UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Fabrício Julian Carini Montenegro

REDES NEURAIS COMO ALTERNATIVAS AO JACOBIANO NA SOLUÇÃO ITERATIVA DA CINEMÁTICA INVERSA

Santa Maria, RS
2019

Fabrício Julian Carini Montenegro

REDES NEURAIS COMO ALTERNATIVAS AO JACOBIANO NA SOLUÇÃO ITERATIVA DA CINEMÁTICA INVERSA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Modelos Analíticos e de Simulação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

ORIENTADOR: Prof. Giovani Rubert Librelotto

COORIENTADOR: Prof. Rodrigo da Silva Guerra

Santa Maria, RS
2019

Carini Montenegro, Fabrício Julian
Redes Neurais Como Alternativas ao Jacobiano na Solução Iterativa da Cinemática Inversa / Fabrício Julian Carini Montenegro.- 2019.
89 p.; 30 cm

Orientador: Giovani Rubert Librelotto
Coorientador: Rodrigo da Silva Guerra
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Ciência da Computação , RS, 2019

1. Robótica 2. Redes Neurais 3. Jacobiano 4. Cinemática Diferencial 5. Cinemática Inversa I. Rubert Librelotto, Giovani II. da Silva Guerra, Rodrigo III. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

©2019
Todos os direitos autorais reservados a Fabrício Julian Carini Montenegro. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
End. Eletr.: fabriciojcmontenegro@gmail.com

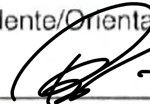
Fabrício Julian Carini Montenegro

REDES NEURAIS COMO ALTERNATIVAS AO JACOBIANO NA SOLUÇÃO ITERATIVA DA CINEMÁTICA INVERSA

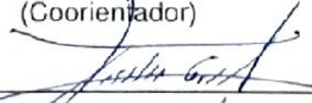
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação, Área de Concentração em Modelos Analíticos e de Simulação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de **Mestre em Ciência da Computação**.

Aprovado em 25 de março de 2019:

Giovani Rubert Librelotto, Dr. (UFSM)
(Presidente/Orientador)



Rodrigo da Silva Guerra, Dr. (UFSM) (videoconferência)
(Coorientador)



Daniel Fernando Tello Gamarra, Dr. (UFSM)

Leonel Pablo Tedesco, Dr. (UNISC) (videoconferência)

Santa Maria, RS
2019

DEDICATÓRIA

À minha mãe.

AGRADECIMENTOS

Agradeço à minha família, aos professores, e aos amigos pelo apoio e colaboração que tornaram possível a conclusão deste trabalho.

Agradeço em especial ao colega Ricardo Grado que desenvolveu o projeto ao meu lado e que trabalhou junto, madrugadas adentro, com ou sem café, para que o projeto fosse encerrado no prazo.

A todos que colaboraram direta ou indiretamente para a conclusão deste processo: obrigado.

All this happened, more or less.

(Kurt Vonnegut - Slaughterhouse-Five)

RESUMO

REDES NEURAIS COMO ALTERNATIVAS AO JACOBIANO NA SOLUÇÃO ITERATIVA DA CINEMÁTICA INVERSA

AUTOR: Fabrício Julian Carini Montenegro
ORIENTADOR: Giovani Rubert Librelotto
COORIENTADOR: Rodrigo da Silva Guerra

A utilização de robôs tornou-se uma prática comum nos dias de hoje graças a grandes avanços tecnológicos que ocorreram nas últimas décadas. Uma das tarefas mais importantes do controle de robôs é o planejamento de seus movimentos no ambiente no qual ele se encontra. Para realizar tarefas no mundo real, as juntas de um robô devem mover-se de forma que seu órgão terminal alcance um determinado objetivo ou siga uma trajetória. O mapeamento desses movimentos feitos pelas juntas para posições no espaço é o problema chamado de cinemática direta, enquanto o problema contrário é chamado de cinemática inversa. O problema da cinemática inversa é geralmente muito complexo e várias soluções tradicionais focam apenas em robôs de topologias específicas. O método iterativo baseado na matriz (pseudo)inversa do Jacobiano é uma abordagem genérica bem conhecida, provada, e confiável que pode ser aplicada a uma variedade de manipuladores. Entretanto, ela depende de linearizações que são válidas somente em uma pequena vizinhança em torno da pose atual do manipulador. Isso exige que o robô mova-se em passos pequenos, recalculando intensamente sua trajetória ao longo do caminho, tornando essa abordagem ineficiente em certas aplicações. Redes neurais, por sua capacidade de modelar sistemas não-lineares, surgem como uma alternativa interessante para a solução deste problema. Neste trabalho é demonstrado que redes neurais realmente podem ser treinadas com sucesso para mapear deslocamentos no espaço de trabalho para incrementos de ângulos de juntas, tendo uma performance melhor que o método baseado na inversa do Jacobiano quando se trata de incrementos de deslocamento maiores e quando em vizinhanças de singularidades. O estudo é validado através de resultados comparativos em manipuladores planares simulados de 2 e 3 graus de liberdade e dá origem a uma abordagem híbrida já aplicada com sucesso em robôs reais.

Palavras-chave: Robótica, Redes Neurais, Jacobiano, Cinemática Diferencial, Cinemática Inversa

ABSTRACT

NEURAL NETWORK AS AN ALTERNATIVE TO THE JACOBIAN FOR ITERATIVE SOLUTION TO INVERSE KINEMATICS

AUTHOR: Fabrício Julian Carini Montenegro
ADVISOR: Giovani Rubert Librelotto
CO-ADVISOR: Rodrigo da Silva Guerra

The usage of robots has become a normal practice today thanks to large technological advances in recent decades. The planning of robots' movements in the environment is one of the most important fundaments in the study of robotics. To complete tasks in the real world, the joints of the robot must move in such a way that the end-effector reach a given goal or follow a trajectory. The mapping from the movements made by the joints to positions in space is the problem called forward kinematics, while the inverse problem is called inverse kinematics. The inverse kinematics problem is generally very complex and many traditional solutions focus only on robots of specific topologies. The iterative method based on the Jacobian's (pseudo)inverse matrix is a well-known, proven, and trusted generic approach that can be applied to many manipulators. However, it depends on linearizations that are valid only on a tight neighborhood around the current pose of the manipulator. This requires the robot to move only in small steps, intensely recalculating its trajectory along the way, making this approach inefficient in some applications. Neural networks, for their capacity of modeling non-linear systems, appear as an interesting alternative to solving this problem. Here, we show that neural networks indeed can be trained successfully to map displacements in the task space to angle increments of the joints, outperforming the method based in the Jacobian's inverse when dealing with larger displacement increments and when near singularities. We validate the study through comparative results in simulated planar manipulators of 2-DOF and 3-DOF and it gives birth to a hybrid approach that has already been succesfully applied in real robots.

Keywords: Robotics, Neural Networks, Jacobian, Differential Kinematics, Inverse Kinematics

LISTA DE FIGURAS

Figura 2.1 – Agentes interagem com o ambiente através de sensores e atuadores. .. 18
Figura 2.2 – Representação simbólica de juntas robóticas. 20
Figura 2.3 – Um mesmo ponto p pode ser representado em dois sistemas de coordenadas diferentes por dois vetores, v_1 e v_2 23
Figura 2.4 – *Roll*, *pitch* e *yaw*. 24
Figura 2.5 – Sistema de coordenadas o_1 em relação a o_0. 25
Figura 2.6 – Representação de p em relação a o_0 e o_1. 26
Figura 2.7 – Visualização de p em relação a o_0. 26
Figura 2.8 – Visualização de p em relação a o_1. 26
Figura 2.9 – Exemplo onde é necessário fazer uma translação e uma rotação. 27
Figura 2.10 – Manipulador planar de duas juntas e valores dos parâmetros de DH. ... 30
Figura 2.11 – Um mesmo braço robótico pode atingir a mesma posição p com diferentes valores de juntas. 31
Figura 2.12 – Modelo matemático de um neurônio. 38
Figura 2.13 – Exemplos de topologias de redes neurais. 39
Figura 2.14 – Da esquerda para a direita, os gráficos das funções de limite rígido, sigmoide, e ReLU. 40
Figura 4.1 – Representação das redes neurais utilizadas para um manipulador de duas e três juntas, respectivamente. 49
Figura 4.2 – Passo 1: Manipulador de 2-DOF em uma pose criada no passo 1, onde θ_1 e θ_2 são ângulos gerados aleatoriamente. 50
Figura 4.3 – Passo 2: Manipulador após o movimento gerado no passo 2, onde $\Delta\theta_1$ e $\Delta\theta_2$ são deslocamentos angulares gerados aleatoriamente. 51
Figura 4.4 – Um mesmo movimento de $45°$ na primeira junta gera resultados diferentes dependendo da pose inicial. No primeiro caso (acima), o braço está estendido. No segundo caso (abaixo), o "cotovelo" está contraído. p indica a posição inicial do *end-effector*, e p' sua posição após o movimento. ... 53
Figura 4.5 – Histogramas mostrando a relação entre o número de amostras e o tamanho dos passos para os intervalos de $0.1mm$ a $20.0mm$ e de $0.1mm$ a $100.0mm$, respectivamente. 54
Figura 4.6 – Histogramas mostrando a relação entre o número de amostras e o tamanho dos passos para os intervalos de $20.0mm$ a $50.0mm$ e de $50.0mm$ a $100.0mm$, respectivamente. 55
Figura 4.7 – Passo 3: p' é a nova posição do *end-effector*, adquirida após o movimento ser aplicado. 56
Figura 4.8 – Passo 4: O vetor Δp é calculado e seu tamanho é analisado para validação. 56
Figura 4.9 – Visualização da quantidade de neurônios por camada da rede neural. .. 59
Figura 5.1 – Poses iniciais dos manipuladores. Linhas pretas representam os segmentos do braço; o círculo azul representa a trajetória. 63
Figura 6.1 – *Traning loss* e *validation loss* das redes neurais treinadas para o braço de três juntas, respectivamente. 65
Figura 6.2 – Braços de duas juntas (2-J) e três juntas (3-J) executando a tarefa no intervalo entre $50mm$ e $100mm$. 66
Figura 6.3 – Comparação do desempenho do método do Jacobiano inverso e da rede

neural quando próximo a uma singularidade. 68

LISTA DE TABELAS

Tabela 2.1 – Parâmetros DH para um manipulador planar de duas juntas, onde θ_1 e θ_2 são variáveis. 29
Tabela 3.1 – Comparação entre os trabalhos relacionados. Os trabalhos são identificados apenas pelo nome do primeiro autor para melhor formatação da tabela. 45
Tabela 4.1 – Entradas e saídas para os braços planares de duas e três juntas. 59
Tabela 5.1 – Valores utilizados para a geração de amostras. 62

LISTA DE ABREVIATURAS E SIGLAS

MSE	*Mean Squared Error* (Erro Quadrático Médio)
DOF	*Degrees of Freedom* (Graus de Liberdade)
DH	Denavit-Hartenberg
$ReLU$	*Rectified Linear Unit* (Unidade Linear Retificada)
RPY	*Roll*, *pitch*, e *yaw*

SUMÁRIO

1 INTRODUÇÃO **13**
2 REVISÃO BIBLIOGRÁFICA **16**
2.1 ROBÓTICA 16
2.1.1 História da Robótica **16**
2.1.2 Definição de Robô e Agente **17**
2.1.3 Tipos de Robô **19**
2.1.4 Tipos de Juntas e Terminologia Robótica **20**
2.2 CINEMÁTICA 21
2.2.1 Descrição Espacial **22**
2.2.1.1 Representação de Posição *23*
2.2.1.2 Representação de Orientação *24*
2.2.1.3 Matriz de Translação *24*
2.2.1.4 Matriz de Rotação *25*
2.2.1.5 Matriz de Transformação Homogênea *27*
2.2.2 Cinemática Direta **28**
2.2.3 Cinemática Inversa **31**
2.2.4 Cinemática de Velocidade e o Jacobiano **32**
2.2.5 Singularidades **35**
2.3 APRENDIZADO DE MÁQUINA 36
2.4 REDES NEURAIS 38
2.5 SUMÁRIO DO CAPÍTULO 41
3 TRABALHOS RELACIONADOS **42**
3.1 COMPARAÇÃO DOS TRABALHOS RELACIONADOS 44
3.2 SUMÁRIO DO CAPÍTULO 46
4 METODOLOGIA **47**
4.1 ENTRADAS E SAÍDAS DA REDE NEURAL 47
4.2 GERAÇÃO DE DADOS 49
4.3 NORMALIZAÇÃO DOS DADOS 57
4.4 ARQUITETURA DA REDE NEURAL 58
4.5 SUMÁRIO DO CAPÍTULO 60
5 CASO DE ESTUDO **61**
5.1 SUMÁRIO DO CAPÍTULO 63
6 RESULTADOS **65**
6.1 SUMÁRIO DO CAPÍTULO 68
7 CONCLUSÃO E TRABALHOS FUTUROS **69**
REFERÊNCIAS BIBLIOGRÁFICAS **70**
ANEXO A – CÓDIGO-FONTE DESENVOLVIDO **72**

1 INTRODUÇÃO

Com os grandes avanços tecnológicos das últimas décadas, a utilização de robôs tornou-se uma prática ubíqua na sociedade moderna. Hoje em dia robôs são amplamente usados na indústria e estão, aos poucos, tornando-se cada vez mais comuns também em ambientes domésticos. Uma das partes mais importantes do controle de robôs, é o planejamento de seus movimentos no ambiente (SPONG et al., 2006; SICILIANO et al., 2010).

Para realizar tarefas no mundo real, geralmente as juntas de um robô devem mover-se de uma maneira tal que o órgão terminal do manipulador, chamado de *end-effector*, alcance um certo objetivo, ou siga uma determinada trajetória. Esse movimento é feito através de movimentos gerados pelos motores que formam as juntas do robô. O mapeamento de posições de juntas para posições no espaço de trabalho é estudado em cinemática direta, enquanto o mapeamento inverso é estudado em cinemática inversa (SPONG et al., 2006).

Especificamente, o problema da cinemática inversa consiste em determinar valores de juntas que posicionem o *end-effector* na posição e orientação desejadas. A solução do problema da cinemática inversa é de fundamental importância no estudo da robótica (SICILIANO et al., 2010).

O problema da cinemática direta pode ser resolvido com regras simples, geralmente através da multiplicação de cadeias de matrizes de transformação homogêneas. O problema da cinemática inversa, entretanto, é bem mais complexo por várias razões. A cinemática inversa geralmente envolve equações algébricas não-lineares para as quais não existe um um algoritmo que forneça uma solução geral. Além disso, geralmente não se pode encontrar soluções fechadas.

Para algumas configurações de braços robóticos, mais do que um conjunto de valores de juntas podem resultar na mesma posição e orientação do *end-effector*. No caso de manipuladores redundantes, intervalos inteiros de soluções podem ser mapeados para a mesma posição e orientação no espaço de trabalho, isto é, existe um conjunto infinito de soluções (SICILIANO et al., 2010).

Além disso, por causa da estrutura cinemática do manipulador, existem casos que não possuem soluções admissíveis. Particularmente, se a posição e orientação do *end-effector* estiverem fora do espaço de trabalho do robô.

Uma das alternativas mais populares para a solução da cinemática inversa, é computar iterativamente a inversa do Jacobiano do robô a cada incremento de pose ao longo da sua trajetória. O Jacobiano é uma função em forma de matriz e pode ser interpretada como a versão de vetor da operação de diferenciação, podendo ser calculado para cada pose do robô.

Os componentes da matriz Jacobiana são derivadas parciais que, para uma dada pose do robô, relacionam a taxa de variação dos valores de juntas à taxa de variação correspondente na posição e orientação do *end-effector.* Através dessa ferramenta, pode-se utilizar sua matriz inversa para computar pequenas variações nos valores de juntas que resultem no pequenos ajustes incrementais desejados para a posição e orientação do *end-effector* de maneira que ele se mova, passo-a-passo, até o objetivo.

Entretanto, pela natureza não-linear do problema da cinemática inversa, o Jacobiano aproxima o comportamento do sistema apenas para uma pequena vizinhança da configuração atual dos valores das juntas. Como consequência, o processo de utilização da inversa do Jacobiano requer que o manipulador aproxime-se do objetivo incrementalmente, executando movimentos muito pequenos por várias iterações.

Outra complicação é que o cálculo da matriz inversa pode ser uma tarefa computacionalmente intensa. Além disso, os incrementos sucessivos podem levar a pose do robô a trajetórias próximas de singularidades, causando instabilidades numéricas muito grandes.

Enquanto isso, percebemos nos últimos anos um renascimento no uso de redes neurais, assim como o surgimento de abordagens de *deep learning* (RUSSELL; NORVIG, 2016). Hoje redes neurais são aplicadas a uma vasta gama de aplicações em vários campos diferentes, de reconhecimento de gestos a reconhecimento de fala, de visão computacional a processamento de linguagem natural. Vários *frameworks* estão disponíveis para o treinamento e aplicação de redes neurais, além de uma grande variedade de dispositivos de *hardware* personalizados especificamente para o seu desenvolvimento, tornando-as ferramentas muito poderosas e versáteis para uma ampla variedade de aplicações computacionais.

Neste trabalho é apresentado um estudo que tem como objetivo geral utilizar redes neurais como uma alternativa para o método Jacobiano, como uma solução iterativa para o problema da cinemática inversa.

O objetivo específico é comparar o desempenho do Jacobiano com um modelo de redes neurais na solução da cinemática inversa, validando a comparação através do erro médio quadrático encontrado durante a execução de uma tarefa específica. A tarefa é utilizar braços robóticos planares simulados de duas e três juntas para descrever uma determinada trajetória circular no espaço, incrementalmente aumentando o passo a ser dado pelo braço e analisando o desempenho das duas técnicas.

O projeto apresentado neste trabalho deu origem a um artigo intitulado *Neural Network as an Alternative to the Jacobian for Iterative Solution to Inverse Kinematics* que foi publicado em novembro de 2018 no Simpósio Latino-Americano de Robótica (LARS) (MONTENEGRO et al., 2018).

O trabalho está organizado da seguinte maneira: no Capítulo 2 são apresentados conceitos importantes para a pesquisa, mais especificamente sobre robótica e redes neurais; no Capítulo 3 são apresentados trabalhos relacionados à pesquisa que oferecem

conhecimentos relevantes ao projeto; o Capítulo 4 descreve a metodologia desenvolvida no trabalho; o Capítulo 5 descreve o ambiente de testes utilizado para a validação da metodologia, enquanto o Capítulo 6 apresenta os resultados dos experimentos e o Capítulo 7 apresenta uma conclusão sobre o trabalho desenvolvido.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo serão abordados temas de relevância para o trabalho. Os temas abordados nesse trabalho podem ser agrupados em duas grandes áreas: Robótica e Aprendizado de Máquina. Este capítulo está dividido em quatro seções, as duas primeiras falando sobre robótica, e as duas últimas falando sobre aprendizado de máquina. Em ambos os casos, uma seção fala sobre o tema de uma maneira abrangente enquanto a outra seção aborda de maneira mais aprofundada os conceitos necessários para a compreensão do trabalho. Desta forma, na próxima seção será abordado o tema robótica.

2.1 ROBÓTICA

Nesta seção é apresentada uma breve história da robótica e, em seguida, são discutidas definições importantes dentro do campo. Mais adiante serão discutidos os diferentes tipos de robôs e definida a terminologia importante para a área.

2.1.1 História da Robótica

Desde a antiguidade a humanidade conta histórias de seres mecânicos que operam de maneira automática. Textos da China antiga contam histórias que aconteceram por volta do ano 1000 A.C. a respeito de um engenheiro chamado Yan Shi que teria apresentado ao rei uma máquina humanoide capaz de se mover e cantar de maneira automática (NEEDHAM; RONAN, 1995). Mesmo que a história seja apenas uma lenda, o sonho de seres mecânicos que podem funcionar de maneira autônoma já existia desde a antiguidade, aparecendo também na cultura grega em obras como a Ilíada de Homero e a lenda de Talos, um gigante feito de bronze que funcionava de maneira automática (SICILIANO et al., 2010).

Através dos anos, robôs se tornaram cada vez mais presentes na cultura humana, inclusive em obras de ficção do século XX, como na peça de Karel Čapek, que introduziu o termo “robô” como é usado hoje (SICILIANO et al., 2010; ORT, 2013; SPONG et al., 2006), e nas histórias de Isaac Asimov, a quem é atribuída a criação do termo “robótica” (ASIMOV; SILVERBERG, 1995; ASIMOV, 1983).

Robôs estão presentes na antiguidade não apenas em obras de ficção, mas também em evidências que sugerem a existência de máquinas reais em locais como a antiga cidade de Alexandria (SHARKEY, 2007). Essas máquinas antigas, conhecidas como *Automatons*, funcionavam de maneira automática e eram projetadas para seguir uma sequência

pré-estabelecida de operações e responder a instruções pré-determinadas.

Nos tempos modernos, a ficção científica influenciou a forma com a qual a população em geral imagina o robô, que ainda o vê como um ser humanoide que pode falar, andar, ver, e ouvir, com uma aparência muito parecida com aquela apresentada por robôs do filme *Metropolis*, um precursor da cinematografia moderna a respeito de robôs. Robôs também são representados dessa forma em exemplos de filmes proeminentes como *Star Wars* e *The Bicentennial Man*, sendo este segundo inspirado em histórias de Asimov (SICILIANO et al., 2010). Diferentemente dessa visão — e apesar da dificuldade e falta de consenso quanto à definição do que pode-se chamar de robô — Spong et al. (2006) afirma que virtualmente qualquer coisa que opera com algum grau de autonomia, geralmente sob o controle de um computador, pode, em algum ponto, ser chamado de robô. A subseção 2.1.2 discute a definição do termo “robô” em mais profundidade.

Com o surgimento da computação digital, o desenvolvimento de robôs acelerou, e em 1961 foi criado o primeiro robô industrial, que trabalhava na linha de montagem da General Motors (NOF, 1999; SICILIANO et al., 2010). Desde então, o desenvolvimento da tecnologia na área de robótica ganhou força e tem acontecido rapidamente. O estudo da robótica tem crescido tremendamente nas últimas décadas, impulsionado pelos avanços acelerados na tecnologia de sensores e computadores e seu consequente barateamento, bem como por avanços teóricos em controle e visão computacional (SPONG et al., 2006).

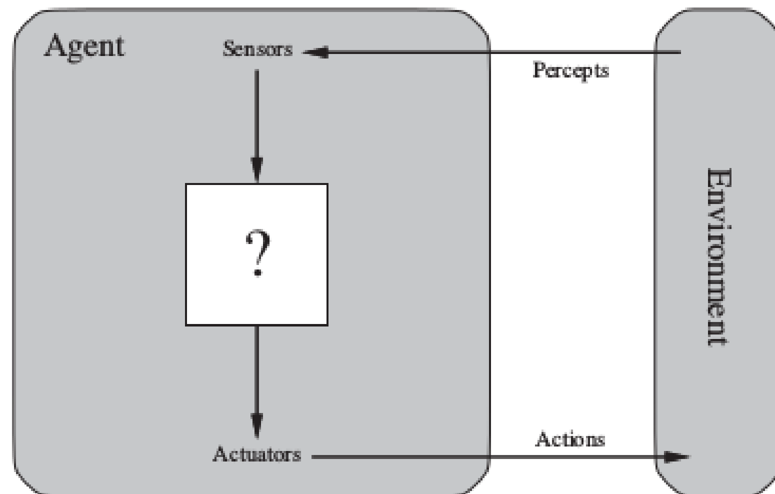
A história da robótica está intimamente ligada à história da humanidade e, por ser um tema tão pervasivo na realidade humana, nota-se sua a importância e a relevância no mundo moderno.

2.1.2 Definição de Robô e Agente

De acordo com estudo científico da robótica, um robô é visto como uma máquina que, independentemente de seu exterior, é capaz de modificar o ambiente no qual ela opera (SICILIANO et al., 2010; RUSSELL; NORVIG, 2016). Uma definição importante nesse ponto é a de agente, que é simplesmente qualquer coisa que pode perceber seu ambiente através de sensores e agir nesse ambiente através de atuadores (RUSSELL; NORVIG, 2016). Um robô pode ser visto como um agente que percebe e atua em seu ambiente. A Figura 2.1 mostra de maneira esquemática como agentes interagem com o ambiente.

Um agente humano, por exemplo, possui olhos ouvidos e outros órgãos que agem como sensores; e mãos, pernas, cordas vocais, etc., que agem como atuadores. Robôs são agentes físicos que executam tarefas através da manipulação de objetos do mundo físico. Para tal, eles são equipados com atuadores como pernas, rodas, juntas e garras. Atuadores têm um único propósito: exercer forças físicas para mover membros ou compo-

Figura 2.1 – Agentes interagem com o ambiente através de sensores e atuadores.



Fonte: Russell e Norvig (2016).

nentes do robô. Estes movimentos podem ter diversos objetivos como locomoção, manipulação, ou a execução de gestos para comunicação. Robôs também são equipados com sensores, que os permitem perceber seu ambiente. A robótica de hoje em dia emprega um conjunto diverso de sensores, incluindo câmeras e lasers para analisar o ambiente, e giroscópios e acelerômetros para analisar o movimento do próprio robô (RUSSELL; NORVIG, 2016).

Desta forma, a definição de agente e, por consequência, a definição de sensores e atuadores é de grande importância na robótica. De fato, a robótica é comumente definida como a ciência que estuda a conexão inteligente entre percepção e ação (SICILIANO et al., 2010).

De maneira geral, o estudo da mecânica e do controle de manipuladores robóticos não é uma ciência nova, mas apenas uma coleção de tópicos presentes em campos mais clássicos. A engenharia mecânica contribui com metodologias para o estudo de máquinas em situações estáticas e dinâmicas. A matemática fornece ferramentas para descrever movimentos espaciais e outros atributos de manipuladores. A teoria de controle fornece ferramentas para projetar e avaliar algoritmos para realizar movimentos desejados ou aplicações de forças. As técnicas de engenharia elétrica são trazidas para ajudar no projeto de sensores e interfaces para robôs, e a ciência da computação contribui como uma base para programar esses dispositivos de maneira que eles possam executar uma determinada tarefa. (CRAIG, 2005).

Desta maneira, pode-se reconhecer que a robótica é um tema interdisciplinar que envolve as áreas de mecânica, controle, computadores, e eletrônica (SICILIANO et al., 2010).

Após definir-se o conceito de robô, é útil ainda classificá-los em tipos, conforme é

explicado na próxima subseção.

2.1.3 Tipos de Robô

Robôs podem apresentar diversas configurações diferentes, e podem ser classificados de acordo com vários critérios, e.g., área de atuação, fonte de energia, geometria, estrutura cinemática, ou método de controle (SPONG et al., 2006). Um critério comum para a classificação de robôs é sua estrutura mecânica.

Analisando esse critério, a maioria dos robôs de hoje em dia encaixam-se em uma de três categorias primárias: manipuladores, robôs móveis, ou a combinação da mobilidade com a manipulação. comumente chamados de manipuladores móveis (RUSSELL; NORVIG, 2016).

Manipuladores, ou braços robóticos, são fisicamente ancorados ao seu espaço de trabalho, i.e., sua base é fixa (SICILIANO et al., 2010; RUSSELL; NORVIG, 2016). Nessa categoria encaixam-se robôs que trabalham, por exemplo, em linhas de montagem ou na Estação Espacial Internacional. Robôs dessa categoria são os mais comuns, contabilizando milhões de unidades instaladas pelo mundo todo (RUSSELL; NORVIG, 2016).

Robôs móveis podem mover-se em seu ambiente utilizando rodas, pernas, ou mecanismos semelhantes. Os robôs utilizados pela NASA na exploração de Marte, *Spirit* e *Opportunity* da missão *Mars Exploration Rover*, e *Curiosity*, ainda ativo, se encaixam nessa categoria (RUSSELL; NORVIG, 2016). *Drones*, apesar de não necessariamente possuírem rodas ou pernas, são considerados robôs aéreos e também se encaixam nessa categoria de robôs.

Manipuladores móveis podem utilizar seus atuadores em espaços mais diversos, mas seu controle é mais difícil por não terem a rigidez que uma base fixa fornece. Robôs humanoides, que imitam o torso humano, se encaixam nessa categoria (RUSSELL; NORVIG, 2016).

Para este trabalho foi utilizado um manipulador fixo, que será tratado na maior parte do texto como braço robótico. Conforme a descrição dada anteriormente, este braço poderia ser anexado a um robô móvel, com rodas ou pernas, e funcionar como um manipulador móvel. Entretanto, este trabalho foca no movimento de apenas um manipulador.

Para colaborar com uma melhor compreensão da metodologia desenvolvida, a próxima subseção apresenta a terminologia relevante para o trabalho.

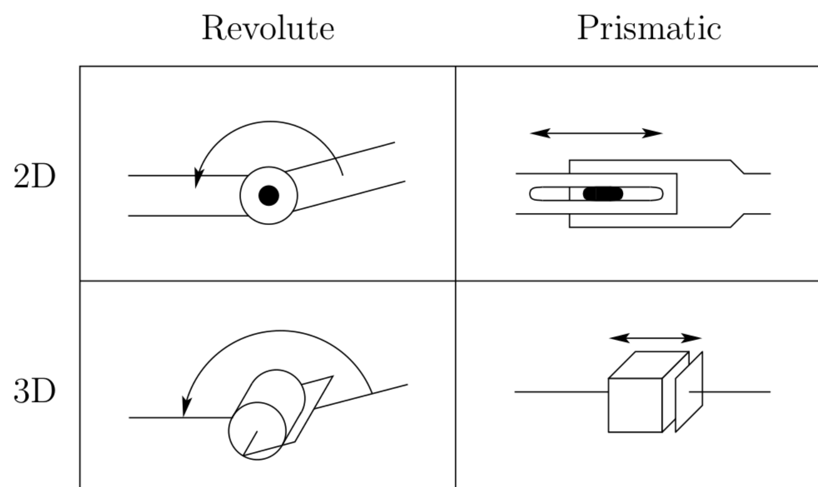
2.1.4 Tipos de Juntas e Terminologia Robótica

Em todas as aplicações de robôs, executar uma determinada tarefa requer a reprodução de um movimento específico planejado pelo robô. Em geral, o problema consiste em criar um sistema de controle que forneça aos atuadores do robô os comandos consistentes com o movimento desejado, o que requer uma análise das características da estrutura mecânica, atuadores, e sensores do robô (SICILIANO et al., 2010; RUSSELL; NORVIG, 2016).

O primeiro problema encontrado é descrever tanto a posição inicial quanto a posição final do atuador relativo a um sistema de coordenadas comum. Para encontrar a solução desse problema, é preciso fazer a modelagem cinemática do manipulador. A formulação das relações cinemáticas dos atuadores permitem o estudo de dois problemas chave em robótica: o problema da cinemática direta e o problema da cinemática inversa (SPONG et al., 2006; SICILIANO et al., 2010). Ambos os problemas são descritos em mais detalhe mais adiante no texto.

Para que a modelagem possa ser feita de maneira satisfatória, é preciso formalizar a simbologia usada para representar robôs. Manipuladores robóticos são compostos de corpos rígidos (*links* ou segmentos) interconectados por articulações (juntas) para formar uma cadeia cinemática (SICILIANO et al., 2010; RUSSELL; NORVIG, 2016). Existem dois tipos básicos de juntas utilizadas em um manipulador: juntas de revolução e juntas prismáticas. Uma junta de revolução funciona como uma dobradiça e permite que aconteça uma rotação relativa entre dois segmentos. Uma junta prismática permite um movimento linear relativo entre dois segmentos. A Figura 2.2 mostra a representação simbólica de juntas robóticas.

Figura 2.2 – Representação simbólica de juntas robóticas.



Fonte: Spong et al. (2006).

O manipulador usado neste trabalho, possui apenas juntas de revolução. As juntas de revolução controlam os ângulos relativos entre os segmentos do braço robótico. Os valores destes ângulos são muito importantes e, no decorrer do texto, serão tratados pelo termo "valores de juntas", "variáveis de juntas", ou simplesmente "ângulos das juntas".

O órgão terminal de um braço robótico geralmente contém uma garra ou outro tipo de ferramenta útil para tarefa a ser executada. Por esse motivo, é comum que o ponto de maior foco no planejamento de um movimento esteja na posição e orientação do órgão terminal do braço. A literatura da área abundantemente chama esse órgão de *end-effector*, ou seja, o a ferramenta acoplada ao final do braço que atua no ambiente. O ambiente de trabalho de um robô é comumente chamado de *workspace* do robô.

Para melhor entender o design de atuadores, é relevante utilizar o conceito de graus de liberdade (*degrees of freedom* ou DOF). Um novo DOF é contado para cada direção independente na qual o robô, ou um de seus atuadores, pode se mover (RUSSELL; NORVIG, 2016). Tipicamente, um manipulador possui pelo menos seis DOF independentes: três para o posicionamento, e três para a orientação (SPONG et al., 2006).

As representações de posição e orientação de um atuador são abordadas com mais profundidade nas Subseções 2.2.1.1 e 2.2.1.2, respectivamente. Com o objetivo de isolar o problema e simplificar a execução de testes para validação, a abordagem utilizada neste trabalho possui foco, primeiramente, em braços robóticos com dois e três DOF, mas ainda é aplicável em braços com mais DOF, conforme é explicado no Capítulo 5.

A próxima seção introduz representações importantes para, em seguida, apresentar um problema fundamental dentro da robótica: o problema da cinemática direta e inversa.

2.2 CINEMÁTICA

Para uma melhor visualização dos problemas da cinemática direta e inversa, é dado o exemplo de uma aplicação básica de robótica: um braço mecânico em uma indústria. O órgão terminal desse braço robótico pode portar uma ferramenta útil para a aplicação, como uma ferramenta de solda, ou um jato de ar para resfriamento, ou até mesmo uma garra para manipulação de objetos. Seja qual for a ferramenta no *end-effector* do braço, os motores que formam as articulações desse braço são os responsáveis por determinar a posição e a orientação dessa ferramenta no espaço de trabalho. Para isso, então, precisamos responder à pergunta: qual a posição resultante da ferramenta quando é dado um conjunto de valores de juntas do braço?

Este é o problema da cinemática direta. De maneira resumida, o problema trata de determinar a posição e a orientação do *end-effector* ou de uma ferramenta em termos das variáveis das juntas (SICILIANO et al., 2010; SPONG et al., 2006). Ou seja, dadas as posições dos motores que formam as juntas de um braço robótico, qual a posição e

orientação de seu órgão terminal?

Mais comum do que este problema, é o problema inverso. Em aplicações reais, como no exemplo de um braço robótico em uma indústria, a posição onde deseja-se que a ferramenta seja colocada é conhecida, esta é a posição alvo do *end-effector*. Em um manipulador, por exemplo, com uma garra robótica no órgão terminal do braço, existe uma posição conhecida de um objeto que deseja-se manipular. Neste caso, é preciso descobrir quais ângulos devem ser aplicados aos motores de maneira que o *end-effector* atinja a posição destino.

A pergunta que segue, portanto, é inversa a do problema anterior. Ou seja, dada uma posição e orientação onde deseja-se que o *end-effector* esteja, quais valores devem ser aplicados às variáveis que controlam os motores para que o braço posicione a ferramenta na posição desejada? Este é o problema da cinemática inversa.

Devido à alta complexidade do problema da cinemática inversa, conforme demonstrado mais adiante, uma outra forma de solução é dada onde o foco é na relação entre as velocidades das juntas e as velocidades lineares e angulares do *end-effector* no *workspace*.

Nas subseções a seguir serão descritos alguns métodos para solucionar os problemas apresentados até o momento e as ferramentas necessárias para aplicar-se estes métodos.

2.2.1 Descrição Espacial

Manipulação robótica, por definição, implica que partes e ferramentas sejam movidas no espaço por algum tipo de mecanismo. Isso naturalmente leva a uma necessidade de representar posições e orientações de partes, ferramentas, e dos mecanismos em si. Para definir e manipular quantidades matemáticas que representam posição e orientação, deve-se definir sistemas de coordenadas e desenvolver convenções para a representação (CRAIG, 2005).

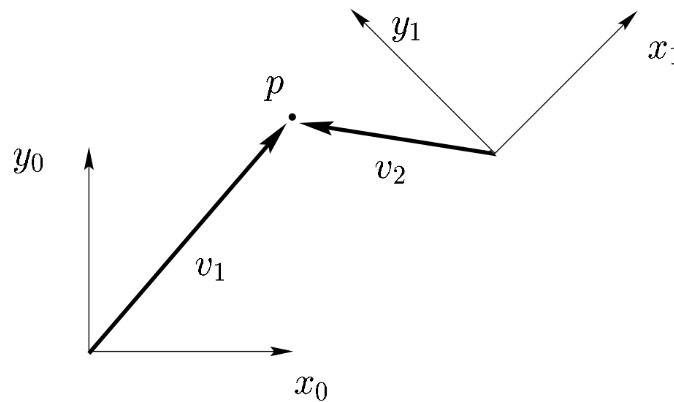
Grande parte do estudo da cinemática de robôs trata de estabelecer vários sistemas de coordenadas para representar as posições e orientações de objetos e transformações entre esses sistemas. A geometria do espaço tridimensional e de movimentos tem um papel central em todos os aspectos da manipulação robótica (SPONG et al., 2006). Existem várias formas de representar posições (coordenadas cartesianas, coordenadas polares, etc) e orientações (*RPY*, ângulos de Euler, etc).

Nas próximas subseções serão apresentadas as representações de posição e orientação utilizadas neste trabalho, bem como matrizes de transformação de fundamental importância para o trabalho aqui apresentado.

2.2.1.1 Representação de Posição

Para apresentar o conceito de representação de posição, utilizará-se um exemplo dado por Spong et al. (2006). Considerando primeiramente o espaço bidimensional, um dado ponto p pode ser representado em um sistema de coordenadas através de coordenadas x e y, ou através de um vetor v_1. Mas esse mesmo ponto pode estar em uma outra posição em relação a outro sistema de coordenadas, o que resulta em um vetor v_2 diferente do primeiro. A representação do ponto p pode ser visualizada na Figura 2.3

Figura 2.3 – Um mesmo ponto p pode ser representado em dois sistemas de coordenadas diferentes por dois vetores, v_1 e v_2



Fonte: Spong et al. (2006).

As coordenadas do ponto p podem ser representadas tanto em relação ao sistema de coordenadas $o_0x_0y_0$ quanto ao sistema $o_1x_1y_1$. Esses valores poderiam ser especificados como:

$$p^0 = \begin{bmatrix} 5 \\ 6 \end{bmatrix}, p^1 = \begin{bmatrix} -3 \\ 4 \end{bmatrix}, \tag{2.1}$$

Geometricamente, o ponto p corresponde a uma posição específica no espaço, mas que pode ser representado de maneiras diferentes através de p^0 e p^1. Uma vez que a origem de um sistema de coordenadas é apenas um ponto no espaço, podemos designar coordenadas que representam a posição da origem em um sistema de coordenadas em relação ao outro. Por exemplo:

$$o_1^0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}, o_0^1 = \begin{bmatrix} -10 \\ -5 \end{bmatrix}, \tag{2.2}$$

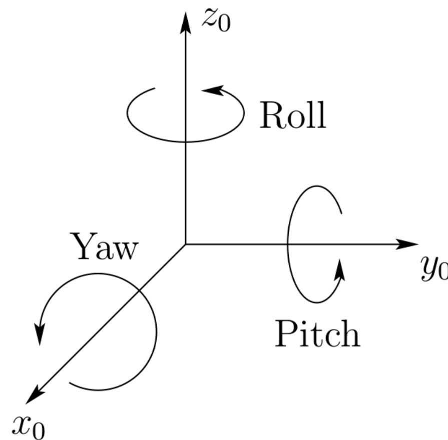
Para executar manipulações algébricas utilizando coordenadas, é essencial que todos os vetores estejam definidos em relação ao mesmo sistema de coordenadas. Dessa forma, fica clara a necessidade de um mecanismo que permita a transformação de coorde-

nadas de pontos que estão expressos em um sistema de coordenadas para coordenadas em relação a algum outro sistema. Essas transformações tipicamente tomam a forma de matrizes de transformação, que são explicadas nas próximas subseções.

2.2.1.2 Representação de Orientação

A representação da orientação pode ser descrita como um produto de sucessivas rotações ao redor dos eixos x, y, e z do sistema de coordenadas. Essas rotações definem os ângulos de *roll*, *pitch*, e *yaw*, que também podem ser descritos como ϕ, θ e ψ, e que são exibidos na Figura 2.4.

Figura 2.4 – *Roll*, *pitch* e *yaw*.



Fonte: Spong et al. (2006).

Para fazer as transformações necessárias entre um sistema de coordenadas e outro, e assim representar a posição e orientação de um dado ponto em relação a outro sistema de coordenadas, é preciso utilizar ferramentas algébricas. Essas ferramentas tomam a forma de matrizes de transformação que, quando multiplicadas pelo ponto em questão, resultam no ponto representado em um outro sistema de coordenadas.

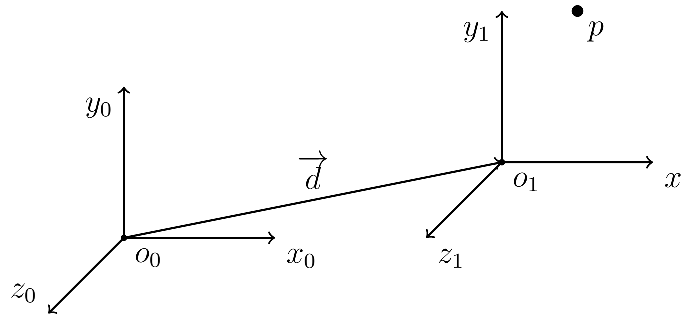
A seguir, são explicadas as matrizes de translação, de rotação, e a matriz de transformação homogênea.

2.2.1.3 Matriz de Translação

A translação move um ponto no espaço por uma distância finita na direção de um dado vetor de deslocamento (CRAIG, 2005).

A posição da origem do sistema de coordenadas o_1 no espaço 3D, quando representada em relação a o_0, i.e., o_1^0, equivale ao vetor de deslocamento $\overrightarrow{d}$ que parte da origem de o_0 e vai até o_1, conforme ilustrado na Figura 2.5. A Figura 2.5 ainda mostra um ponto p em uma determinada posição no espaço.

Figura 2.5 – Sistema de coordenadas o_1 em relação a o_0.



Fonte: Próprio autor.

É dado que mesmo ponto p possui uma representação p^0 no primeiro sistema de coordenadas, e p^1 no segundo sistema. Suponha que $p^1 = \begin{bmatrix} 1 & 2 & 0 \end{bmatrix}^T$, temos como tarefa encontrar o valor de p^0. Suponha ainda que $\overrightarrow{d} = \begin{bmatrix} 5 & 1 & 0 \end{bmatrix}^T$ representa a translação que leva da origem do primeiro sistema à origem do segundo, ou seja, a posição relativa da origem do segundo sistema, descrita em termos das coordenadas do primeiro sistema. Com essas informações podemos encontrar p^0 através da soma da matriz de translação T que, neste cenário, trata-se do valor de $\overrightarrow{d}$. Sendo assim, podemos utilizar a matriz de translação $T = \begin{bmatrix} d_x & d_y & d_z \end{bmatrix}^T = \begin{bmatrix} 5 & 1 & 0 \end{bmatrix}^T$ que, somada ao ponto p^1, resulta no ponto p^0.

$$p^0 = p^1 + T = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 5 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \\ 0 \end{bmatrix} \tag{2.3}$$

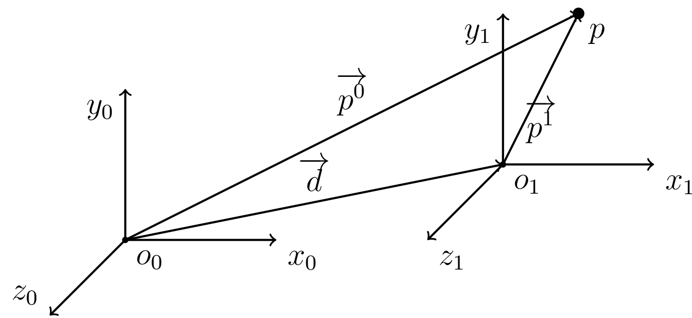
Este processo pode ser visualizado na Figura 2.6.

Além da transformação de translação, outra transformação importante é a de rotação, que será apresentada na subseção seguinte.

2.2.1.4 Matriz de Rotação

Para atingir a orientação desejada, muitas vezes é necessário rotacionar um sistema de coordenadas. É dado um sistema de coordenadas o_0, e um sistema o_1 que está na mesma posição, mas com seus eixos rotacionados $90°$ ao redor do eixo z (movimento

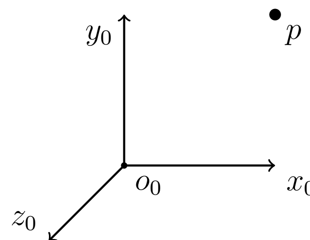
Figura 2.6 – Representação de p em relação a o_0 e o_1.



Fonte: Próprio autor.

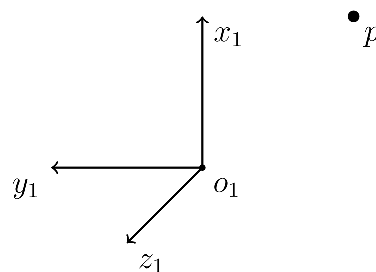
que seria chamado de *roll*). É dado ainda um ponto p em uma posição no espaço que, apesar da diferença na orientação dos eixos de coordenadas, ainda está na mesma posição. As Figuras 2.7 e 2.8 ilustram ambos os sistemas de coordenadas e onde o ponto p se encontra em relação a cada um.

Figura 2.7 – Visualização de p em relação a o_0.



Fonte: Próprio autor.

Figura 2.8 – Visualização de p em relação a o_1.



Fonte: Próprio autor.

O formato da matriz de rotação ao redor do eixo z é:

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

Multiplicando o ponto $p^1 = \begin{bmatrix} 2 & -2 & 0 \end{bmatrix}^T$ pela matriz de rotação R_z, para $\theta = 90°$, obtemos o valor de p^0.

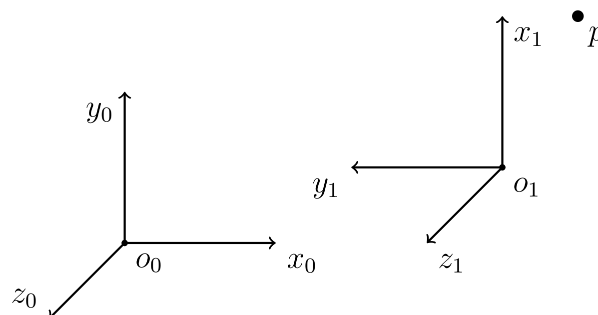
$$p^0 = R_z(\theta)p^1 = \begin{bmatrix} cos(90°) & -sin(90°) & 0 \\ sin(90°) & cos(90°) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 0 \end{bmatrix} \quad (2.5)$$

Matrizes de translação e de rotação são ferramentas muito importantes, e com essas ferramentas em mãos, é possível criar uma ferramenta ainda mais poderosa, a matriz de transformação homogênea, que será abordada na próxima subseção.

2.2.1.5 Matriz de Transformação Homogênea

Em inúmeras situações, é necessário fazer tanto uma translação quanto uma rotação. É dado um exemplo na Figura 2.9 onde $p^1 = \begin{bmatrix} 2 & -1 & 0 \end{bmatrix}^T$, o sistema o_1 está rotacionado em $90°$, e a distância entre os dois sistemas é dada pelo vetor $\overrightarrow{d} = \begin{bmatrix} 5 & 1 & 0 \end{bmatrix}^T$.

Figura 2.9 – Exemplo onde é necessário fazer uma translação e uma rotação.



Fonte: Próprio autor.

Neste caso, uma ferramenta de extrema utilidade é a matriz de transformação homogênea, que contém tanto as transformações de translação quanto as transformações de rotação em apenas uma matriz. Para este exemplo, considerando o vetor de deslocamento $\overrightarrow{d}$ e a matriz de rotação no eixo z, R_z, a matriz de transformação homogênea teria

o seguinte formato:

$$H = \begin{bmatrix} cos\theta & -sin\theta & 0 & d_x \\ sin\theta & cos\theta & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.6)$$

De fato, multiplicando a matriz H pelo ponto p^1, obtemos o ponto p_0, i.e., $p^0 = Hp^1$. Neste caso, é adicionada uma quarta linha com o valor 1 ao ponto p^1 para obter-se sua representação homogênea, ou seja, $p^1 = \begin{bmatrix} p^1_x & p^1_y & p^1_z & 1 \end{bmatrix}^T$. Executando a multiplicação temos:

$$Hp^1 = \begin{bmatrix} cos\theta & -sen\theta & 0 & d_x \\ sen\theta & cos\theta & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p^1_x \\ p^1_y \\ p^1_z \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta.p^1_x - sen\theta.p^1_y + d_x \\ sen\theta.p^1_x + cos\theta.p^1_y + d_y \\ p^1_z + d_z \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 3 \\ 0 \\ 1 \end{bmatrix} \quad (2.7)$$

Essas são ferramentas úteis para a solução do problema da cinemática direta, que será apresentado na próxima subseção.

2.2.2 Cinemática Direta

O problema da cinemática direta se preocupa com a relação entre as juntas individuais do robô e a posição e orientação da ferramenta ou *end-effector*. De maneira mais formal, o problema da cinemática direta é determinar a posição e orientação do *end-effector*, dados os valores para as variáveis das juntas do robô. As variáveis das juntas são os ângulos entre os segmentos no caso de juntas rotacionais ou de revolução, e a extensão do segmentos no caso de juntas prismáticas. O problema da cinemática direta contrasta com o problema da cinemática inversa, que será abordado na próxima subseção, e que trata de determinar valores para os ângulos das juntas de forma que se alcance uma posição e orientação desejada para o *end-effector* de um robô (SPONG et al., 2006).

Boa parte do estudo da cinemática de robôs se concentra em estabelecer vários sistemas de coordenadas para representar as posições e orientações de objetos rígidos e encontrar transformações entre esses sistemas de coordenadas (SPONG et al., 2006). Essa parte do estudo engloba boa parte das disciplinas de geometria espacial e álgebra linear, levando em consideração o estudo de vetores, transformações entre espaços vetoriais e o uso de matrizes como representação dessas informações.

A representação de espaços vetoriais em cada uma das juntas e as transformações de pontos de um desses espaços vetoriais são conhecimentos relevantes para a manipu-

lação de variáveis para o cálculo da solução do problema. Para encontrar a posição de um objeto relativa à base do braço e representar o mesmo ponto relativo ao órgão terminal do braço, por exemplo, matrizes de transformação homogêneas são usadas.

Dessa forma, com o conhecimento obtido no uso de matrizes de transformação, é possível descobrir a posição do *end-effector* de um robô em relação ao seu sistema de coordenadas base simplesmente aplicando matrizes de transformação. A construção dessas matrizes pode ser uma tarefa não trivial e, por essa razão, algumas técnicas existem para padronizar o processo de construção das mesmas.

Uma técnica amplamente usada é conhecida como a representação de Denavit-Hartenberg (DH) (DENAVIT; HARTENBERG, 1955). Essa abordagem se preocupa em atribuir sistemas de coordenadas a cada junta de uma maneira padronizada, de forma que quatro atributos possam ser avaliados. Com esses atributos em mãos, a convenção DH oferece uma maneira confiável de adquirir as matrizes de transformação de um segmento do braço para outro e, consequentemente, a matriz de transformação resultante que nos permite identificar a posição do órgão terminal do braço em relação à sua base.

Além dessa abordagem, existem abordagens geométricas que se baseiam na observação do sistema e na extração de fórmulas a partir do conhecimento matemático. Pode-se, por exemplo, encontrar relações entre os ângulos aplicados às juntas e a posição final do *end-effector* através de triângulos e relações trigonométricas encontradas na configuração do braço. Essa abordagem consiste em observar a forma com que os segmentos do braço se encaixam e determinar relações entre equações conhecidas para determinar novos valores. Esse processo é difícil de ser automatizado, já que a pessoa usa seu conhecimento prévio de geometria e álgebra para encontrar formas e fórmulas geométricas que se encaixem no problema, o que não é um problema trivial.

Por fins de ilustração, é dado um exemplo onde o cálculo é feito a partir dos parâmetros DH de um manipulador planar de duas juntas. Por concisão, o processo usado para encontrar os parâmetros DH não serão discutidos aqui. Considerando o braço ilustrado na Figura 2.10, os parâmetros DH encontrados seriam os exibidos na Tabela 2.1.

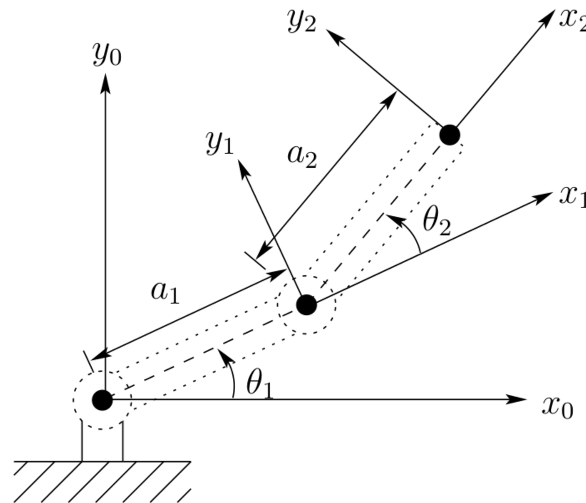
Tabela 2.1 – Parâmetros DH para um manipulador planar de duas juntas, onde θ_1 e θ_2 são variáveis.

Link	a_i	α_i	d_i	θ_i
1	a_1	0	0	θ_1
2	a_2	0	0	θ_2

Fonte: Adaptado de Spong et al. (2006)

O sistema o_0 é estabelecido como base. Uma vez que a base está estabelecida, os sistemas de coordenadas o_1 e o_2 são fixados conforme a convenção de DH. Note que o_2 se encontra exatamente no ponto que representa o *end-effector*. Neste caso, a tarefa é encontrar a matriz T_0^2 que encontra a posição de o_2 em relação a o_0. Conforme mostrado

Figura 2.10 – Manipulador planar de duas juntas e valores dos parâmetros de DH.



Fonte: Spong et al. (2006).

na figura 2.10, os parâmetros de DH nos dão valores a serem utilizados nas matrizes de transformação. A matriz final T_0^2 então é:

$$T_0^2 = \begin{bmatrix} cos(\theta_1+\theta_2) & -sen(\theta_1+\theta_2) & 0 & a_1cos(\theta_1)+a_2cos(\theta_1+\theta_2) \\ sen(\theta_1+\theta_2) & cos(\theta_1+\theta_2) & 0 & a_1cos(\theta_1)+a_2sen(\theta_1+\theta_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

Considerando que o ponto p que deseja-se encontrar é $p = T_0^2 o_2$ temos:

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_1sen(\theta_1)+a_2cos(\theta_1+\theta_2) \\ a_1sen(\theta_1)+a_2sen(\theta_1+\theta_2) \\ 0 \end{bmatrix} \tag{2.9}$$

Assim, os valores de x e y podem ser resolvidos de maneira simples através da substituição dos valores da Tabela 2.1 nas equações.

É relevante notar que o processo da cinemática direta, principalmente quando aplicado em um ambiente 2D, pode ser trivial. É possível encontrar a posição do *end-effector* em relação à base do robô apenas aplicando relações trigonométricas. Quando a situação for mais complexa, com braços robóticos com mais juntas em ambientes 3D, o uso de matrizes homogêneas facilita o processo.

Na próxima subseção será abordado o problema inverso.

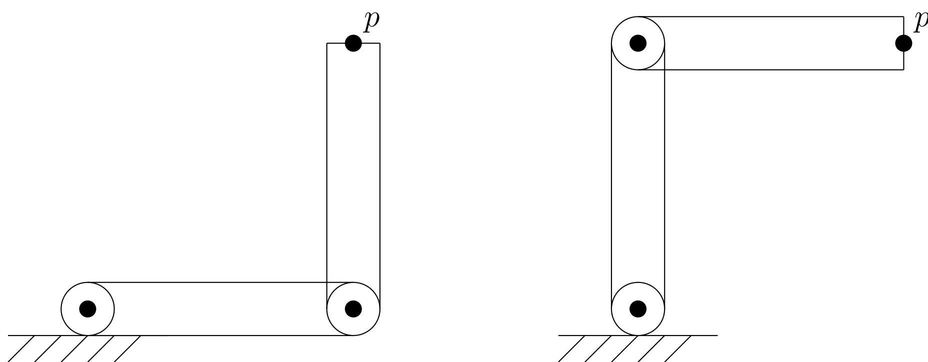
2.2.3 Cinemática Inversa

A cinemática inversa é o problema inverso ao problema da cinemática direta. Enquanto a cinemática direta tenta determinar a posição e orientação do *end-effector* em termos das variáveis das juntas, o problema da cinemática inversa tenta encontrar as variáveis das juntas em termos da posição e orientação do *end-effector* (SPONG et al., 2006). Este problema é, em geral, muito mais difícil do que o problema da cinemática direta.

Partindo do problema da cinemática direta, uma vez que foram encontradas as fórmulas matemáticas que descrevem a posição do *end-effector* em termos de suas juntas, devemos então inverter as funções, representando os valores das juntas em termos da posição do *end-effector*. Essa pode ser uma tarefa árdua já que as funções resultantes do estudo da cinemática direta raramente são funções lineares. Conforme visto no exemplo da subseção anterior, mesmo utilizando-se a abordagem de Denavit-Hartenberg para obter uma matriz de transformação confiável, é difícil encontrar elementos dentro da matriz que possam ser invertidos de maneira simples. Os termos geralmente estão "escondidos" dentro de funções trigonométricas, o que torna a inversão das funções uma tarefa não trivial.

É importante manter em mente que braços robóticos possuem uma área limitada onde eles conseguem trabalhar, chamada *workspace*. Quando o ponto desejado se encontra dentro dessa área, geralmente existem diferentes soluções para o problema, uma vez que o braço pode posicionar-se de maneiras diferentes e ainda assim manter o *end-effector* no mesmo ponto, conforme ilustrado pela Figura 2.11.

Figura 2.11 – Um mesmo braço robótico pode atingir a mesma posição p com diferentes valores de juntas.



Fonte: Próprio autor.

Neste caso, o sistema de equações que descrevem o problema da cinemática inversa para um dado robô possui múltiplas soluções. Já no caso de o ponto desejado estar na borda do *workspace* do robô, o problema terá apenas uma solução. Pontos que exigem que o robô estenda totalmente o braço, por exemplo, se encaixam nesse caso. Uma última

possibilidade é a de que o ponto desejado esteja fora do *workspace* do robô. Nesse caso, as equações não terão solução, já que o ponto desejado não pode ser alcançado pelo braço.

Essas diferentes possibilidades de solução tornam o problema da cinemática inversa ainda mais complexo visto que, dependendo da posição do alvo, o problema pode possuir uma, múltiplas ou nenhuma solução para um mesmo braço robótico.

Tomando em consideração todos esses fatores, Siciliano et al. (2010) lista de maneira sucinta motivos pelos quais, em geral, o problema da cinemática inversa é muito mais complicado do que o problema da cinemática direta.

- As equações para solução, em geral, não são lineares e, portanto, nem sempre é possível encontrar uma solução na forma fechada
- Múltiplas soluções podem existir
- Infinitas soluções podem existir, e.g., no caso de um manipulador cinematicamente redundante
- Pode ser que não existam soluções admissíveis, dependendo da estrutura do manipulador cinemático

Por esses motivos, uma alternativa interessante é o uso da cinemática de velocidades, através do Jacobiano, que será abordada na próxima subseção.

2.2.4 Cinemática de Velocidade e o Jacobiano

Matematicamente, as equações da cinemática direta definem a função entre o espaço cartesiano de posições e orientações e o espaço de valores de juntas. As relações de velocidade são determinadas pelo Jacobiano dessa função. O Jacobiano é uma função em forma de matriz e pode ser interpretada como a versão de vetor de uma derivada comum de uma função escalar. Esse Jacobiano, ou matriz Jacobiana, é uma das quantidades mais importantes na análise e controle do movimento de robôs. Ela surge em virtualmente todos os aspectos da manipulação robótica: no planejamento e execução de trajetórias suaves, na determinação de singularidades, na execução de movimentos coordenados antropomórficos, na derivação de equações dinâmicas de movimento, e na transformação de forças e torques do *end-effector* para as juntas do manipulador (SPONG et al., 2006). Na cinemática de velocidade, ou cinemática diferencial, estudamos a relação entre as velocidades de variação dos valores de juntas e da posição e orientação do *end-effector*.

No caso de movimentos do *end-effector*, podemos analisar dois tipos de velocidade: velocidade angular e velocidade linear. A velocidade angular analisa a velocidade com a

qual a orientação do *end-effector* muda, e a velocidade linear analisa a velocidade com a qual a posição do *end-effector* muda.

Para encontrarmos o operador Jacobiano, considere o manipulador com n juntas e com variáveis de juntas q_1,...,q_n. Temos que

$$T_n^0(q) = \begin{bmatrix} R_n^0(q) & o_n^0(q) \\ 0 & 1 \end{bmatrix} \tag{2.10}$$

representa a transformação do sistema de coordenadas do *end-effector* para o sistema de coordenadas da base, onde $q = (q_1,...,q_n)^T$ é o vetor de variáveis das juntas. Conforme o robô se move, ambas as variáveis q_i, e a posição o_n^0 e orientação R_n^0 do *end-effector* serão funções do tempo. O objetivo é relacionar a velocidade linear e angular do *end-effector* ao vetor de velocidades de juntas $\dot{q}(t)$.

Considerando que ω_n^0 é o vetor de velocidade angular do *end-effector*, e que v_n^0 é o vetor de velocidade linear do *end-effector*, são procuradas expressões do tipo

$$v_n^0 = Jv\dot{q} \tag{2.11}$$

$$\omega_n^0 = J\omega\dot{q} \tag{2.12}$$

onde Jv e $J\omega$ são matrizes $3 \times n$. É possível escrever as Equações 2.11 e 2.12 juntas como

$$\begin{bmatrix} v_n^0 \\ \omega_n^0 \end{bmatrix} = J_n^0\dot{q} \tag{2.13}$$

onde J_n^0 é dado por

$$J_n^0 = \begin{bmatrix} Jv | J\omega \end{bmatrix} \tag{2.14}$$

.

A matriz J_n^0 é chamada de Jacobiano do Manipulador ou simplesmente Jacobiano. Note que J_n^0 é uma matriz $6 \times n$ onde n é o número de juntas.

Como exemplo, levando em consideração a velocidade linear Jv, o valor a ser analisado é o quanto a posição do *end-effector* varia em relação a uma dada variação nos valores das juntas. Considerando um braço de três juntas, A, B, e C, o interesse se dá na variação das componentes x, y, e z (posição do *end-effector*) em relação à variação nos ângulos dessas juntas. Para esse caso, a matriz Jacobiana J (que não leva em considera-

ção a orientação do *end-effector*, de acordo com o foco deste trabalho) seria:

$$J = \begin{bmatrix} \partial p_x/\partial\theta_A & \partial p_x/\partial\theta_B & \partial p_x/\partial\theta_C \\ \partial p_y/\partial\theta_A & \partial p_y/\partial\theta_B & \partial p_y/\partial\theta_C \\ \partial p_z/\partial\theta_A & \partial p_z/\partial\theta_B & \partial p_z/\partial\theta_C \end{bmatrix} \tag{2.15}$$

A variação na posição p do *end-effector*, representada por $\dot{p}$, é dada por

$$\dot{p} = J(q)\dot{q} \tag{2.16}$$

É importante notar que nesta equação estão os valores da variação nas juntas $\dot{q}$ e o interesse está em descobrir a variação resultante $\dot{p}$. Este é o problema da cinemática direta, onde têm-se os valores das juntas e deseja-se descobrir a posição do *end-effector*. Para o caso inverso, é informada a variação $\dot{p}$ que deseja-se aplicar no *end-effector*, e precisa-se descobrir qual a variação $\dot{q}$ nos valores das juntas que deve-se aplicar para atingir o movimento desejado. Dessa forma, tem-se que

$$\dot{q} = J(q)^{-1}\dot{p} \tag{2.17}$$

Note que o cálculo dessa equação exige que seja feita a inversão de uma matriz, um processo que é computacionalmente custoso quando se tratando de um robô com vários DOFs. Além disso, o valor da derivada só é válido para um pequeno intervalo.

Em cálculo, a derivada em um ponto de uma função $y = f(x)$ representa a taxa de variação instantânea de y em relação a x neste ponto. Geometricamente, a derivada no ponto $x = a$ de $y = f(x)$ representa a inclinação da reta tangente ao gráfico desta função no ponto $(a, f(a))$. Pela definição de derivada, analisam-se dois valores de x infinitamente próximos, ou seja, a e $a + h$, onde h tende infinitamente a zero.

De maneira similar, o Jacobiano relaciona a variação instantânea da posição p do *end-effector* em relação a uma variação da pose q. De fato, uma variação infinitamente pequena nos valores de juntas q irá gerar um movimento que descreve uma linha reta, exatamente como modelado pelo Jacobiano. Isso significa que quanto menores os intervalos de variação utilizados, mais preciso será o cálculo. Por esse motivo, um movimento preciso requer que o cálculo seja repetido a cada pequeno passo. Isso acaba exigindo um poder computacional muito grande, uma vez que para cada pequeno intervalo todo o processo deve ser repetido.

No caso do exemplo dado, existem três juntas e três coordenadas do espaço 3D. Mas, ao utilizar um manipulador com o mesmo número de juntas em uma situação em um espaço 2D, ou seja, considerando um braço planar de três juntas, percebe-se que a matriz J resultante não será uma matriz quadrada, uma vez que terá três colunas e apenas duas linhas.

Um manipulador que está equipado com mais DOF internos do que são necessá-

rios para uma dada tarefa, e.g., um manipulador de três juntas que deve posicionar-se no plano, é considerado um manipulador redundante. Para estes casos, não existem soluções únicas para o problema da cinemática inversa (SPONG et al., 2006). Além disso, a matriz Jacobiana de um manipulador redundante não é quadrada e, portanto, não pode ser invertida para resolver o problema da velocidade inversa.

Para tratar desse problema, é utilizado o conceito da pseudoinversa, uma aproximação matemática que acaba carregando erros em seu valor.

Ou seja, apesar de ser uma boa alternativa para a cinemática inversa, a solução da cinemática inversa de velocidades através do uso do Jacobiano contém problemas próprios, como a exigência de um alto poder computacional por sua natureza iterativa, e complicações na inversão da matriz como no caso de manipuladores redundantes.

Além disso, mais um fator potencialmente complicante, é a existência de singularidades, que será abordada na próxima subseção.

2.2.5 Singularidades

Em matemática, uma singularidade é geralmente um ponto no qual um dado objeto matemático não é definido, ou um ponto de um conjunto excepcional onde ele não se comporta corretamente de alguma maneira específica, como a possibilidade de diferenciação. Por exemplo, a função $f(x) = \frac{1}{x}$ na linha dos reais tem uma singularidade em $x = 0$, onde ela "explode" para $\pm\infty$ e não está definida (BERRESFORD; ROCKETT, 2015).

O Jacobiano é uma função de q e, por esse motivo, valores de q para os quais a ordem de J diminui possuem uma significância especial. De maneira similar à definição matemática, as configurações nas quais isso acontece são chamadas singularidades. Identificar as singularidades de um manipulador é importante por várias razões, listadas por Spong et al. (2006).

1. Singularidades representam configurações a partir das quais certas direções de movimento podem não ser possíveis.

2. Em singularidades, velocidades limitadas do *end-effector* podem corresponder a velocidades ilimitadas (infinitas) das juntas.

3. Em singularidades, forças e torques limitados do *end-effector* podem corresponder a torques ilimitados de juntas.

4. Singularidades geralmente (mas não sempre) correspondem a pontos na fronteira do *workspace* do manipulador, i.e., correspondem a pontos no limite de alcance do manipulador.

5. Singularidades correspondem a pontos no *workspace* do manipulador que podem ser inalcançáveis sob pequenas perturbações nos parâmetros dos segmentos, tais como comprimento e deslocamento.

6. Próximos a singularidades não existem soluções únicas para o problema da cinemática inversa. Nesses casos pode não haver uma solução ou existirem infinitas soluções.

Esse é um dos fatores negativos do uso do Jacobiano. Para contornar esse e outros problemas, listados na subseção anterior, é possível usar técnicas de aprendizado de máquina, que não necessariamente se baseiam nas mesmas representações matemáticas para a solução do problema. Para alguns algoritmos de aprendizado de máquina, como redes neurais artificiais, problemas como a inversão de matrizes e instabilidade quando próximo a singularidades não se aplicam. Isso faz com que possa-se utilizar o potencial do método Jacobiano como alternativa para o problema da cinemática inversa, evitando seus problemas.

Neste trabalho, em específico, a técnica de aprendizado de máquina escolhida foram as redes neurais artificiais. O tópico de aprendizado de máquina é abordado na próxima seção.

2.3 APRENDIZADO DE MÁQUINA

Aprendizado de máquina é um campo da inteligência artificial que usa técnicas estatísticas para dar a sistemas de computador a habilidade de aprender a partir de dados, sem ser explicitamente programado (KOZA et al., 1996).

Um agente está aprendendo se ele melhora sua performance em tarefas futuras após fazer observações sobre o mundo. Russell e Norvig (2016) afirmam que existem três razões principais para almejar que agentes possam aprender. Primeiro, os projetistas não podem antecipar todas as possíveis situações nas quais um agente irá encontrar-se. Por exemplo, um robô projetado para navegar em labirintos deve aprender o *layout* de cada novo labirinto que ele encontra. Segundo, os projetistas não podem antecipar todas as mudanças que acontecem com o passar do tempo; um programa projetado para prever os valores de amanhã da bolsa de valores deve aprender a adaptar-se quando as condições mudam. Terceiro, às vezes os próprios programadores humanos não têm nenhuma ideia de como programar a solução. Por exemplo, a maioria das pessoas é boa em reconhecer os rostos de membros da família, mas mesmo os melhores programadores são incapazes de programar um computador que execute essa tarefa, exceto usando algoritmos de aprendizagem.

Existem diferentes formas de aprendizado que, ainda segundo Russell e Norvig (2016), dependem de quatro fatores principais:

- Quais componentes podem ser melhorados.
- Qual conhecimento o agente já possui.
- Qual a representação usada para os dados e o componente.
- Qual *feedback* está disponível para ser usado no aprendizado.

Existem três tipos de *feedback* que determinam três tipos principais de aprendizado.

No aprendizado não supervisionado, o agente aprende os padrões contidos nas entradas mesmo que nenhum *feedback* explícito seja fornecido. A técnica mais comum de aprendizado não supervisionado é *clustering*, que detecta grupos potencialmente úteis de exemplos de entrada.

No aprendizado por reforço, o agente aprende a partir de uma série de reforços na forma de recompensas ou punições. Cabe ao agente aprender quais ações prévias ao reforço foram as mais responsáveis por ele.

E no aprendizado supervisionado, o agente observa alguns pares entrada–saída dados como exemplo e aprende a função que mapeia a entrada para a saída.

A distinção entre os tipos de aprendizado de máquina nem sempre é tão clara, mas pode-se afirmar que a técnica usada neste trabalho se encaixa no tipo de aprendizado supervisionado.

Redes neurais são ferramentas muito poderosas e versáteis, que podem ser aplicadas aos mais diversos problemas. Um redescobrimento da técnica nos últimos anos trouxe uma grande disponibilidade de ferramentas e *frameworks* para treinar e aplicar redes neurais, além de uma grande quantidade de *hardwares* especializados estarem amplamente disponíveis. Esse redescobrimento é devido, em grande parte, ao desenvolvimento de técnicas de *deep learning*, que hoje são usadas para resolver problemas que sobreviveram por muitos anos, mesmo com os esforços da comunidade de inteligência artificial (LECUN; BENGIO; HINTON, 2015).

Para o problema da cinemática inversa, especificamente, redes neurais surgem como uma alternativa atraente por serem capazes de modelar complexos problemas não-lineares. Isso se dá principalmente por causa das funções de ativação utilizadas no núcleo dos neurônios, conforme será apresentado na próxima seção.

Redes neurais possuem uma facilidade de modelagem, provida pela ampla disponibilidade de ferramentas e *frameworks*; possuem um grande poder de representação, sendo aplicadas nos mais diferentes campos; e podem ser utilizadas para modelar problemas não-lineares. Por esses motivos, redes neurais artificiais foram a técnica escolhida para este trabalho.

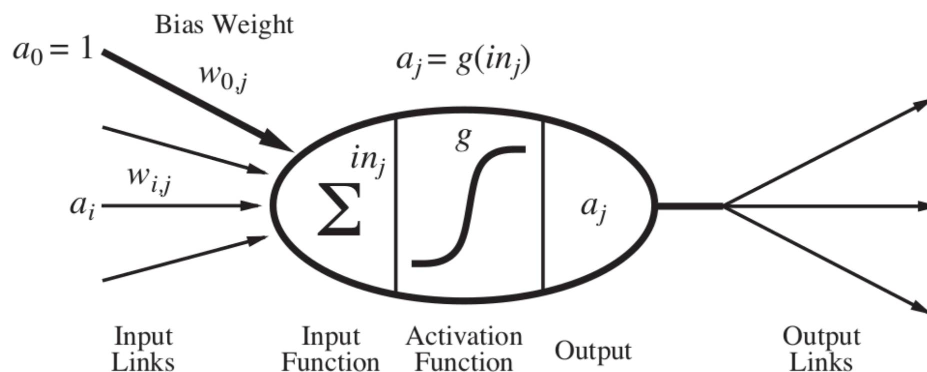
A próxima seção discute em mais profundidade o funcionamento de redes neurais artificiais.

2.4 REDES NEURAIS

As ideias discutidas em técnicas de aprendizado de máquina são úteis para criar modelos matemáticos sobre a atividade do cérebro humano. De maneira inversa, pensar sobre o cérebro se provou útil no passado para estender o escopo das ideias técnicas (RUSSELL; NORVIG, 2016).

Avanços em neurociência levaram à descoberta de que a atividade cerebral consiste primariamente de atividade eletroquímica em redes de células cerebrais chamadas neurônios. Inspirados por esse conhecimento, alguns dos trabalhos mais antigos de inteligência artificial tinham como objetivo criar modelos que funcionassem como redes neurais artificiais, e o objetivo de criar modelos de sistemas biológicos ainda é bem comum nos dias de hoje. A Figura 2.12 mostra um simples modelo matemático de um neurônio criado por McCulloch e Pitts (1943). De maneira simplificada, o neurônio ativa quando uma combinação linear de suas entradas excede algum limitante, i.e., o neurônio implementa um classificador linear.

Figura 2.12 – Modelo matemático de um neurônio.

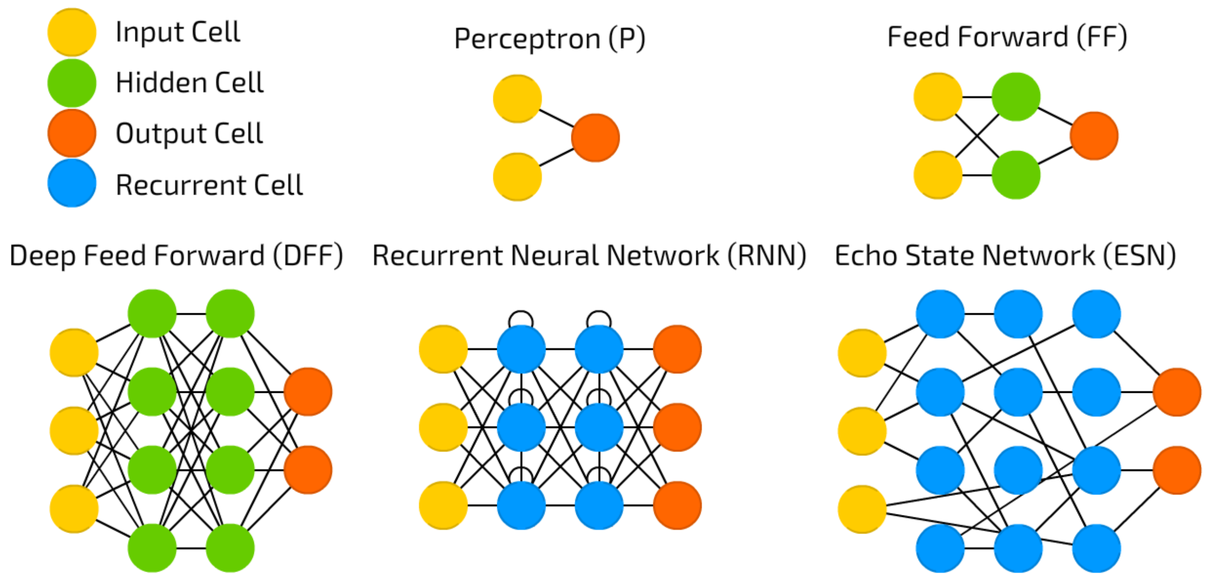


Fonte: Russell e Norvig (2016).

Uma rede neural não é nada mais do que uma coleção de neurônios conectados. As propriedades de uma rede são determinadas por sua topologia — i.e. pela maneira com a qual neurônios se conectam — e pelas propriedades dos neurônios. A Figura 2.13 mostra exemplos de diferentes topologias de redes neurais.

Conforme a definição de Russell e Norvig (2016), redes neurais são compostas por neurônios conectados por sinapses. Uma sinapse de um neurônio i a um neurônio j serve para propagar a ativação a_i de i para j. Cada sinapse também possui um peso

Figura 2.13 – Exemplos de topologias de redes neurais.



Fonte: Adaptado de Tch (2017).

numérico $w_{i,j}$ associado a si, que determina a força e o sinal da conexão. Assim como em modelos de regressão linear, cada neurônio possui uma entrada "falsa" $a_0 = 1$ com um peso associado $w_{0,j}$. Esta entrada "falsa" é chamada *bias*. Cada neurônio j primeiro computa a soma ponderada de suas entradas:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i \tag{2.18}$$

A função de ativação g então é aplicada a essa soma, e isso resulta na saída:

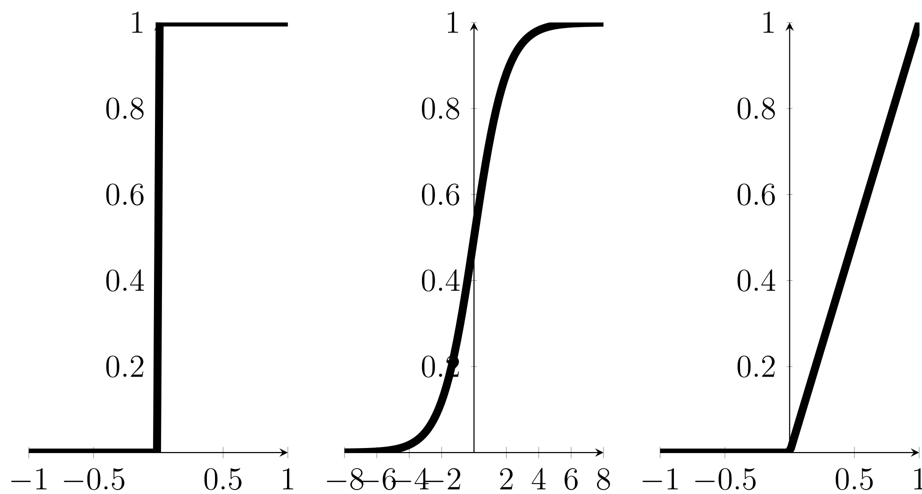
$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{2.19}$$

Durante muito tempo, a função de ativação g utilizada era um limitante rígido ou uma função sigmoide, mas hoje em dia a função conhecida como ReLU (Rectified Linear Unit) é a mais popular. A função de limite rígido possui saída apenas igual a 0 ou 1 enquanto a função logística, também conhecida como função sigmoide por sua forma de "S", possui uma variação mais suave. Já a função ReLU, é igual a 0 quando $x < 0$ e é uma função linear para os demais casos. As três funções são ilustradas na Figura 2.14.

Em todos os casos, essas funções de ativação não lineares garantem a propriedade importante de que a rede neural como um todo pode representar funções não lineares, como o problema da cinemática inversa, que é um problema altamente não linear.

Neurônios são conectados a neurônios adjacentes formando camadas que são,

Figura 2.14 – Da esquerda para a direita, os gráficos das funções de limite rígido, sigmoide, e ReLU.



Fonte: Próprio autor.

normalmente, separadas em três categorias, a camada de entrada, a camada de saída, e as camadas escondidas. A camada de entrada contém neurônios através dos quais o sistema é alimentado, enquanto a camada de saída contém neurônios cuja saída será também considerada a saída do sistema. As camadas que recebem sinapses da camada de entrada e que conectam-se entre si até chegar na camada de saída são chamadas camadas escondidas, mas a topologia de uma rede neural pode ser mais complexa, com conexões entre neurônios da mesma camada, conexões que pulam camadas, etc.

Existem duas formas fundamentalmente distintas de decidir como conectar os neurônios de uma rede neural. Uma rede neural *feed-forward* possui conexões apenas em uma direção, i.e., ela forma um grafo acíclico direcionado. Cada neurônio, também considerado um nó na rede, recebe suas entradas de neurônios anteriores e entrega sua saída para neurônios adiante, não existem ciclos. Uma rede neural *feed-forward* representa a função de sua entrada atual, portanto, não possui estado interno a não ser os pesos em si.

Uma rede neural recorrente, por outro lado, alimenta suas saídas novamente às suas próprias entradas. Isso significa que os níveis de ativação da rede formam um sistema dinâmico que pode alcançar um estado estável ou exibir oscilações ou até mesmo um comportamento caótico. Além disso, a resposta da rede a uma certa entrada depende de seu estado inicial, o que pode depender de entradas anteriores. Dessa forma, redes neurais recorrentes, diferentemente de redes *feed-forward*, podem ter uma memória de curto prazo.

Para os experimentos executados neste trabalho, o comportamento de memória não seria necessário, e por esse motivo a rede neural escolhida foi uma *feed-forward* simples.

A maior complicação do uso de redes neurais é a adição de camadas escondidas

à rede. O erro da camada de saída é claro: basta analisar a diferença Δ entre o valor de saída e o valor que era esperado. Já o erro das camadas escondidas não é tão óbvio porque os dados de treinamento não dizem que valores os nós internos deveriam ter. Felizmente, é possível propagar o erro da camada de saída para as camadas escondidas através do processo chamado *back-propagation*. O processo de *back-propagation* surge diretamente da derivação do gradiente do erro total e pode ser sumarizado da seguinte maneira:

- Computar os valores de Δ para os neurônios de saída, utilizando o erro observado.
- Começando na camada de saída, repita os seguintes passos para cada camada da rede até que a primeira camada escondida seja alcançada:
  - Propague os valores de Δ para a camada anterior.
  - Atualize os pesos entre as duas camadas.

Atualmente, muitos *frameworks* estão disponíveis e implementam todo o processo de criação e treinamento de redes neurais, desde a criação e ligação de neurônios até o processo de *back-propagation*. O uso desses *frameworks* torna o emprego de redes neurais mais viável e uma alternativa interessante para a resolução de problemas das mais variadas formas.

2.5 SUMÁRIO DO CAPÍTULO

O Capítulo 2 apresenta os temas de robótica e aprendizado de máquina, e foca, dentro desses temas, em cinemática e redes neurais. Conforme apresentado na Seção 2.2, o problema da cinemática inversa é complexo e sua solução não é trivial. Por esse motivo, o método que usa o Jacobiano invertido surge como uma boa alternativa. Apesar disso, o Jacobiano possui suas próprias limitações, como a instabilidade para passos longos e quando próximo a singularidades.

Enquanto isso, redes neurais apresentam-se como uma alternativa forte para os mais diversos problemas, inclusive de natureza não-linear, como o problema da cinemática inversa. Através do uso de redes neurais artificiais, pode-se melhorar o desempenho do método Jacobiano em situações nas quais ele pode ser impreciso. Mais do que isso, o uso de redes neurais nesse contexto elimina a preocupação em relação a inversão de matrizes, um processo computacionalmente custoso e que muitas vezes depende de aproximações matemáticas.

O Capítulo 3 descreve outros trabalhos que utilizam redes neurais na área de robótica, e apresenta comparações entre eles e o trabalho aqui apresentado.

3 TRABALHOS RELACIONADOS

Conforme o que foi apresentado no Capítulo 2, fica claro que o problema da cinemática inversa, mesmo quando utilizando a abordagem iterativa através do Jacobiano, é um problema não trivial e com espaço para melhorias. Técnicas de aprendizado de máquina, em especial redes neurais, até mesmo por sua ampla disponibilidade, são ferramentas muito versáteis e poderosas para as mais variadas tarefas que podem ser aplicadas ao problema.

O trabalho aqui apresentado explora o uso de redes neurais como uma alternativa para a abordagem Jacobiana, como uma solução iterativa para o problema da cinemática inversa. Da mesma forma, vários trabalhos abordam esse problema através de técnicas similares.

A maior parte dos trabalhos encontrados que utilizam abordagem semelhantes, usando redes neurais na área da robótica, foca em tentar resolver o problema geral da cinemática inversa, em alguns casos com resultados relativamente bons.

Duka (2014) utiliza uma rede neural para controlar os movimentos de um braço planar no espaço 2D. Em sua abordagem, as entradas da rede neural são as coordenadas x e y do plano e uma orientação θ alvo, e a saída são valores de juntas que o robô deve aplicar para atingir o alvo. Os valores são dados em termos absolutos, e não em termos de deslocamentos em relação à posição atual do *end-effector*. Ou seja, o trabalho utiliza a rede neural *feed-forward*, nesse caso com uma camada de 100 neurônios escondidos, que modela o problema da cinemática inversa. O problema da cinemática inversa possui um espaço não-linear muito grande para ser modelado em uma rede neural simples e, conforme o próprio autor conclui, apesar de os resultados serem promissores, a utilização de uma rede neural para resolver o problema da cinemática inversa requer um estudo mais aprofundado, e existem várias direções que precisam de atenção e que podem ser melhoradas.

Almusawi, Dülger e Kapucu (2016) utilizam uma abordagem semelhante à apresentada por Duka (2014) em um robô com 6 DOF. A diferença na metodologia proposta por Almusawi, Dülger e Kapucu (2016) é a adição dos valores atuais das juntas do robô como entrada da rede neural. Segundo sua conclusão, isso melhora muito o resultado em relação a uma rede neural que utiliza como entrada somente a posição e orientação desejadas para o *end-effector*, recebendo como saída os valores de juntas necessários para atingir o alvo. Ou seja, Almusawi, Dülger e Kapucu (2016) mostram que o uso das posições atuais das juntas do robô melhora o resultado da rede neural.

Neste caso, o próximo passo parece ser o uso da cinemática diferencial inversa que, justamente, utiliza as posições atuais do braço robótico e calcula apenas o deslocamento necessário nas juntas para realizar o movimento desejado no *end-effector*. Essa é

a abordagem apresentada nesse trabalho.

Aggarwal, Aggarwal e Urbanic (2014) também utilizam redes neurais para solucionar o problema da cinemática inversa, utilizando como entrada da rede os valores de x, y e z do alvo, e recebendo na saída os valores de juntas do robô de 6 DOF utilizado no experimento. Após o treinamento e validação da rede treinada, os autores utilizaram um *dataset* contendo pontos de singularidade conhecidos do robô para uma segunda seção de treinamento da rede. O objetivo é ensinar a rede a detectar zonas próximas a singularidades e evitá-las, diferentemente do objetivo do trabalho aqui apresentado, onde queremos obter uma rede neural que controle o braço robótico de maneira robusta mesmo quando próximo a singularidades. A partir de Aggarwal, Aggarwal e Urbanic (2014) ainda aprendemos que essa técnica requer pouco tempo computacional quando comparada a métodos tradicionais.

Além do uso de redes neurais *feedforward* para a solução do problema da cinemática inversa, pode-se encontrar na literatura trabalhos que utilizam redes neurais de maneiras mais complexas para solucionar o problema. KöKer (2013) utiliza uma abordagem híbrida com redes neurais recorrentes em paralelo e usa algoritmos genéticos como método para redução do erro e treinamento das redes. O trabalho conclui que este método gera resultados muito precisos, mas com um alto custo computacional, e menciona que pesquisas mais aprofundadas devem ser feitas para aumentar a velocidade do algoritmo genético e reduzir o tempo total de processamento.

Conforme discutido por Bingul, Ertunc e Oysu (2005) e Oyama et al. (2001), pela natureza do problema da cinemática inversa e por sua complexidade, modelar a cinemática inversa de um robô simples com uma rede neural é uma tarefa difícil mesmo para redes neurais grandes. Por esse motivo, modelar a abordagem Jacobiana em uma rede neural surge como uma opção interessante.

Hasan et al. (2010) utiliza uma abordagem semelhante à apresentada nesse trabalho, utilizando a velocidade angular das juntas como saída da rede neural, mas o treino é feito para uma trajetória específica, i.e., não espera-se que a rede atue em trajetórias diferentes. Foram coletados 600 *datasets* para cada intervalo de 1 segundo do robô utilizado executando o movimento em uma trajetória conhecida – 400 foram utilizados para o treinamento, e 200 para o processo de testes.

Ainda em Hasan et al. (2010), percebe-se que a tarefa da rede neural é aprender a velocidade de execução do movimento. As entradas da rede neural são a posição atual do *end-effector* em x, y e z; a orientação atual utilizando o sistema RPY de representação; e a velocidade linear do *end-effector*, representada em ângulos/segundos. A saída da rede contém a posição angular e a velocidade angular.

No trabalho aqui apresentado, a posição angular é utilizada como entrada, e não existe um controle da velocidade linear, uma vez que deseja-se mover o braço robótico sem necessariamente controlar a velocidade do movimento. A ênfase está em melhorar

a performance do método do Jacobiano inverso como um todo, tornando-o viável para passos grandes e melhorando o desempenho em termos de custo computacional, além de gerar bons resultados quando próximo a singularidades.

3.1 COMPARAÇÃO DOS TRABALHOS RELACIONADOS

Analisando de maneira objetiva os trabalhos relacionados listados acima no texto, percebe-se que Duka (2014), Almusawi, Dülger e Kapucu (2016), Aggarwal, Aggarwal e Urbanic (2014), e KöKer (2013) utilizam redes neurais para resolver o problema da cinemática inversa. Duka (2014) utiliza um braço planar simples e chega à conclusão de que, apesar dos resultados serem promissores, o problema da cinemática inversa é complexo e requer estudos mais aprofundados. Almusawi, Dülger e Kapucu (2016) utiliza um braço de 6 DOF 3D e usa as posições atuais das juntas como entrada, o que melhora os resultados. Aggarwal, Aggarwal e Urbanic (2014) também utiliza um braço de 6 DOF, mas não inclui as posições atuais das juntas como entradas. Além disso, é feito um segundo treinamento para detectar singularidades conhecidas e evitá-las.

KöKer (2013) vai além e utiliza redes neurais recorrentes em paralelo e algoritmos genéticos como método de treinamento. Seus resultados são muito precisos, mas o método possui um custo computacional muito alto, o que não ocorre nos trabalhos que usam apenas redes neurais *feed-forward*.

Em sua análise, Bingul, Ertunc e Oysu (2005) e Oyama et al. (2001) afirmam que o problema da cinemática inversa é muito complexo para uma simples rede neural, e que até mesmo redes neurais maiores podem ter dificuldade em aprender o problema. Isto é, alimentar a rede neural apenas com as posições alvo não é o suficiente para que ela modele o problema de maneira satisfatória.

Conforme Almusawi, Dülger e Kapucu (2016) afirmam, a inclusão das posições atuais das juntas como entrada da rede neural gera resultados melhores. Isso significa que a rede neural modela melhor o problema da cinemática inversa quando ela conhece a posição atual das juntas.

A Tabela 3.1 mostra de forma mais sucinta a comparação entre os trabalhos relacionados mencionados Neste capítulo.

Tabela 3.1 – Comparação entre os trabalhos relacionados. Os trabalhos são identificados apenas pelo nome do primeiro autor para melhor formatação da tabela.

Trabalho	Objetivo	Técnica Usada	Conclusão
Duka[1]	Cinemática Inversa.	Rede Neural com posição e orientação alvo como entrada.	Resultados promissores, mas requer estudos mais aprofundados.
Almusawi[2]	Cinemática Inversa.	Rede Neural incluindo valores de juntas atuais como entrada.	Utilizar a pose atual do robô como entrada melhora o desempenho da rede neural.
Aggarwal[3]	Cinemática Inversa e detecção de zonas próximas a singularidades para que sejam evitadas.	Rede Neural com posição e orientação alvo como entrada.	O uso de redes neurais possui baixo custo computacional.
KöKer[4]	Cinemática Inversa.	Redes Neurais Recorrentes em paralelo e algoritmos genéticos como otimizadores.	Resultados extremamente precisos, mas possui um custo computacional muito alto.
Bingul[5] e Oyama[6]	Comparação de abordagens.	—	Redes neurais simples possuem dificuldades em modelar o problema da cinemática inversa.
Hasan[7]	Controlar a velocidade de movimento em uma trajetória específica.	Rede Neural incluindo velocidade angular como saída.	O deslocamento angular pode ser usado como saída da rede neural com sucesso.

Fonte: Próprio autor.

[1] Duka (2014)
[2] Almusawi, Dülger e Kapucu (2016)
[3] Aggarwal, Aggarwal e Urbanic (2014)
[4] KöKer (2013)
[5] Bingul, Ertunc e Oysu (2005)
[6] Oyama et al. (2001)
[7] Hasan et al. (2010)

3.2 SUMÁRIO DO CAPÍTULO

Analisando a literatura da área, percebe-se que existe um grande potencial no uso de redes neurais para solucionar o problema da cinemática inversa. Através da análise dos trabalhos presentes na literatura, é possível notar que o uso da posição atual das juntas do robô como entrada da rede neural melhora seu desempenho, e que a simples modelagem do problema da cinemática inversa através de uma rede neural gera resultados aceitáveis mas imprecisos.

Não foram encontrados na literatura trabalhos que utilizam redes neurais da maneira como é feita neste trabalho, modelando exatamente o método Jacobiano através de redes neurais artificiais. A utilização da posição atual das juntas como entrada da rede neural parece permitir que ela seja capaz de representar internamente as posições resultantes do *end-effector*, para então receber a variação desejada na posição do *end-effector* e dar como resposta a variação necessária nos valores das juntas.

Esse formato é exatamente o mesmo dado pelas equações da cinemática de velocidade, sem apresentar as limitações matemáticas do método, o que torna essa metodologia uma solução elegante para o problema.

O próximo capítulo explora em profundidade a metodologia desenvolvida a partir do que pode ser observado no estudo dos temas de cinemática e de redes neurais, aplicando noções importantes aprendidas através do estudo de trabalhos relacionados desenvolvidos pela comunidade científica.

4 METODOLOGIA

Na abordagem apresentada nesse trabalho, utilizou-se uma rede neural em substituição à matriz invertida do Jacobiano. O objetivo é obter um modelo de rede neural que funcione bem em situações nas quais a inversa do Jacobiano não funciona. Para esse fim, assume-se que os dados utilizados como entradas e saídas são os mesmos utilizados quando o problema é resolvido através do método Jacobiano.

A validação da metodologia se dá através de um caso de estudo que utiliza braços robóticos simulados, e será apresentada em detalhe no Capítulo 5.

Para esse trabalho, o foco do cálculo é apenas nas coordenadas da posição do *end-effector*, sem levar em consideração a orientação. Isto é, a rede neural modela a velocidade linear, mas não a velocidade angular do *end-effector*. Entretanto, a técnica pode ser expandida para incluir, além da posição, a orientação do *end-effector* e, por consequência, sua velocidade angular, conforme explicado na Seção 4.1.

Além disso, a técnica aqui apresentada pode ser estendida para braços com um maior número de juntas e tanto para manipuladores simulados em espaços 3D quanto para manipuladores no mundo real.

Nas próximas seções serão apresentados detalhes a respeito dos dados utilizados como entradas e saídas da rede neural, a técnica utilizada para a geração dos dados de treinamento utilizados neste trabalho, e a arquitetura da rede neural e seus hiperparâmetros. A descrição de como foram executados os testes usados para validar a metodologia é apresentada no Capítulo 5.

4.1 ENTRADAS E SAÍDAS DA REDE NEURAL

Conforme visto no Capítulo 3, em geral, algoritmos que abordam o problema da cinemática inversa recebem valores absolutos das coordenadas da posição alvo do *end-effector* e calculam os valores de junta absolutos correspondentes que levam o *end-effector* até essa posição alvo.

No caso da cinemática de velocidade, quando se tratando do problema da cinemática direta, as informações dadas são os valores de juntas atuais e uma variação a ser aplicada nesses valores. Como saída da equação, esperamos obter a variação resultante na posição do *end-effector*.

No processo inverso, como é o caso estudado nesse trabalho, são dados os valores de junta atuais do manipulador para o cálculo da matriz Jacobiana, e a variação que deseja-se causar na posição do *end-effector*. Como resultado, deseja-se receber a variação incremental nos valores de juntas correspondentes ao movimento do *end-effector*.

É importante perceber que, como demonstrado na subseção 2.2.4, a matriz Jacobiana J é uma função do conjunto de valores de juntas atuais q. Além disso, pode-se notar na equação 2.17, que o termo que multiplica a matriz Jacobiana inversa é a derivada de tempo da posição do *end-effector* $\dot{p}$.

Para fins práticos, podemos chamar o vetor de valores de juntas de θ ao invés de q, uma vez que ele contém os valores dos ângulos individuais entre cada um dos n segmentos do braço robótico (θ_1, θ_2, ..., θ_n). Ainda podemos chamar a variação na posição p do *end-effector* de Δp, uma vez que estamos analisando apenas um instante de tempo. O resultado da equação é a derivada de tempo dos valores de juntas $\dot{q}$ que, para aplicação, podemos chamar de $\Delta\theta$.

Ou seja, as informações fornecidas para a equação são os valores de juntas θ das juntas rotacionais e também a variação Δp desejada na posição do *end-effector*. Como resposta, espera-se a variação $\Delta\theta$ que deve ser aplicada nas juntas do robô de maneira que se atinja o movimento desejado Δp.

A rede neural utilizada neste trabalho tem como objetivo modelar o funcionamento da matriz $J(\theta)^{-1}$ quando aplicada ao problema da cinemática inversa. Para tanto, as informações fornecidas à rede devem ser as mesmas utilizadas na equação. Desta forma, as entradas da rede neural são:

1. Os valores de juntas θ atuais das juntas rotacionais;

2. A variação Δp desejada da posição do *end-effector*.

Para manipuladores genéricos trabalhando em um espaço 3D, θ teria tantos componentes quanto o número de juntas. Nos casos dos braços de 2-DOF e 3-DOF apresentados nesse trabalho, θ possui, respectivamente, dois e três valores de juntas.

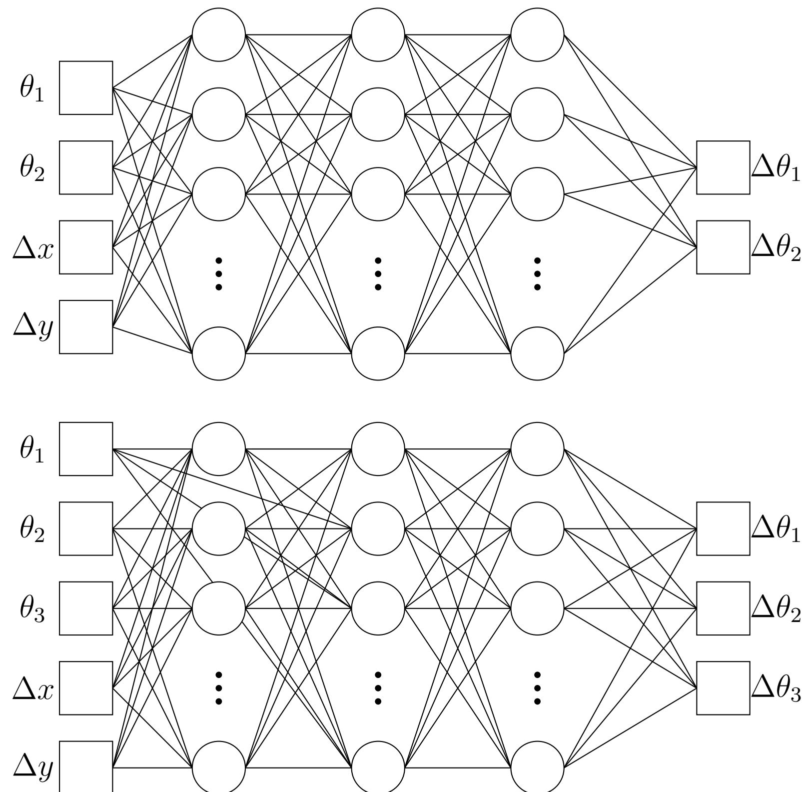
Os valores utilizados nesse trabalho levam em consideração apenas a posição do *end-effector* e não sua orientação. Neste caso, para o braço de 2 DOF, decidimos tomar em consideração dois componentes em Δp (Δx e Δy), enquanto, para o braço de 3 DOF, consideramos três componentes (Δx, Δy e Δz). Além disso, Δp poderia conter até seis valores, se levarmos em consideração os valores de orientação (variação em *roll*, *pitch* e *yaw*).

A saída da rede neural possui tantos componentes quanto forem necessários para representar o vetor $\Delta\theta$ que representa as mudanças incrementais nos valores de juntas que resultam no Δp desejado. Ou seja, um braço com duas juntas possuiria dois componentes na saída da rede ($\Delta\theta_1$ e $\Delta\theta_2$) enquanto um braço com três juntas possuiria três componentes ($\Delta\theta_1$, $\Delta\theta_2$ e $\Delta\theta_3$). Desta forma, a expansão dessa técnica para braços com um maior número de juntas ainda terá tantos componentes na saída quanto o número de juntas.

A Figura 4.1 ilustra as redes neurais utilizadas para manipuladores de duas e três juntas, respectivamente. As entradas e saídas utilizadas para os casos estudados nesse

trabalho são apresentadas no Capítulo 5.

Figura 4.1 – Representação das redes neurais utilizadas para um manipulador de duas e três juntas, respectivamente.



Fonte: Próprio autor.

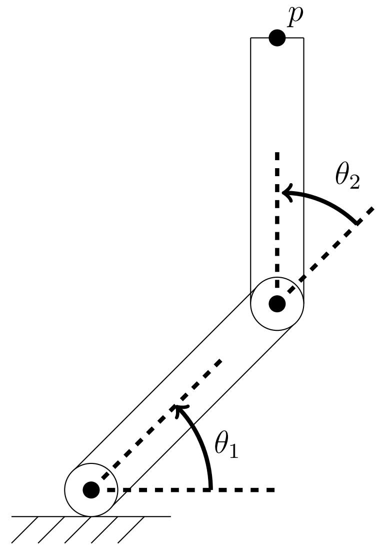
Na próxima seção, o método utilizado para a geração dos dados de treinamento da rede neural é apresentado.

4.2 GERAÇÃO DE DADOS

O treinamento de redes neurais exige um grande número de amostras, que são formadas por pares de entradas e saídas. Para esta abordagem, usou-se a cinemática direta para o processo de geração das amostras individuais. O problema da cinemática direta possui solução mais simples e pode fornecer os dados necessários para o treinamento da rede que modela o Jacobiano. Esse processo se dá através de quatro passos sequenciais.

O primeiro passo é gerar uma pose aleatória válida para o braço robótico. Isso pode ser feito através da geração de valores aleatórios para cada um dos ângulos de juntas contidos em θ. A Figura 4.2, que ilustra o primeiro passo do processo de geração de dados quando aplicado a um braço de 2 DOF.

Figura 4.2 – Passo 1: Manipulador de 2-DOF em uma pose criada no passo 1, onde θ_1 e θ_2 são ângulos gerados aleatoriamente.



Fonte: Próprio autor.

Para simplificar o processo, esse passo significa simplesmente gerar qualquer pose para o manipulador, sem necessariamente checar por auto-colisões. Isso significa que o sistema não avalia quando um segmento do braço colide com outro segmento. Entretanto, deve ser notado que a detecção de colisão é um passo importante do processo de programação de movimentos em robôs reais, uma vez que robôs possuem partes sensíveis e de alto custo que podem ser danificadas em colisões. Por esse motivo, a detecção de colisões deve ser implementada para a aplicação desta técnica em robôs do mundo real. Esta parte do processo foi ignorada pelo fato de que os braços planares de duas e três juntas utilizados como caso de estudo para esta abordagem possuem configurações simples e, neste caso, serão avaliados somente em um ambiente bidimensional simulado.

Para gerar essas posições aleatórias, os valores de juntas assumem ângulos aleatórios que, como consequência, colocam o *end-effector* em uma posição aleatória no espaço.

Os ângulos aleatórios gerados para cada junta do manipulador são retirados de uma distribuição uniforme no intervalo entre $-\pi$ e π. A utilização de uma distribuição uniforme garante que todos os valores entre $-\pi$ e π possuam uma probabilidade igual de serem selecionados. A utilização de valores que cobrem toda a amplitude de movimento de cada junta faz com que as posições resultantes do *end-effector* cubram todo o *workspace* do braço robótico, uma vez que esse é todo o intervalo de valores possíveis para cada junta.

O motivo para o uso desses valores é que cada junta é capaz de cobrir todos os ângulos de uma circunferência completa, isto é, seu ângulo em relação ao sistema de referência pode variar entre 0 e 2π. Entretanto, testes sucessivos mostraram que a utilização

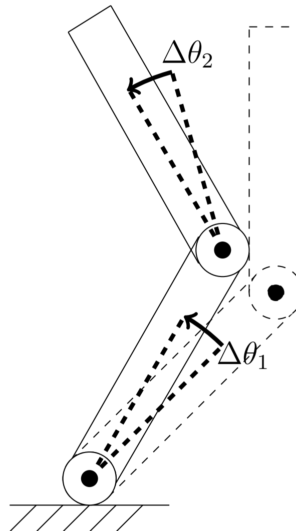
de valores entre $-\pi$ e π gera uma melhor representação a ser modelada pela rede neural, por tratarem-se de quantidades equilibradas entre valores positivos e negativos.

Dessa forma, os valores de cada junta (θ_1, θ_2, ..., θ_n) assumem ângulos aleatórios entre $-\pi$ e π. Esses valores formam o vetor θ que, após ser gerado durante esse primeiro passo, é armazenado para ser normalizado em passos seguintes e então alimentado à rede neural para treinamento. Os valores contidos em θ formam o primeiro conjunto de entradas da rede neural.

Com essa pose inicial aleatória em mãos, é possível aplicar a cinemática direta para encontrar a posição p resultante do *end-effector*. Essa posição p é uma informação importante que também é armazenada para uso mais adiante no processo.

O segundo passo é, partindo de cada uma das poses aleatórias geradas durante o primeiro passo, gerar pequenas mudanças incrementais ($\Delta\theta_1$, $\Delta\theta_2$, ..., $\Delta\theta_n$) nos valores de juntas. Esses valores formarão o conjunto $\Delta\theta$. Para isso, deve-se gerar valores aleatórios que serão usados como deslocamento angular em cada uma das juntas. A Figura 4.3 mostra um manipulador de duas juntas após um movimento aleatório $\Delta\theta$ ser aplicado.

Figura 4.3 – Passo 2: Manipulador após o movimento gerado no passo 2, onde $\Delta\theta_1$ e $\Delta\theta_2$ são deslocamentos angulares gerados aleatoriamente.



Fonte: Próprio autor.

Os movimentos a serem aplicados nas juntas podem ser tanto em sentido horário quanto em sentido anti-horário, ou seja, pode-se aplicar movimentos com valores de ângulos tanto positivos quanto negativos.

Conforme explicado na Seção 2.2.4, o Jacobiano é muito preciso para variações pequenas nos valores de juntas do manipulador, uma vez que o problema se aproxima de uma função linear na vizinhança da pose original. O objetivo da abordagem é obter uma rede neural que também tenha uma boa performance nesses casos. Por esse motivo, é

preciso treiná-la utilizando amostras nas quais o tamanho de Δp seja pequeno o suficiente para comparar sua performance à do Jacobiano. Também deseja-se que a rede neural tenha uma boa performance em intervalos maiores, onde o Jacobiano é impreciso, o que exige um treinamento com amostras nas quais o tamanho de Δp é maior. Isso gera uma necessidade de controlarmos o tamanho do vetor Δp resultante nas amostras geradas, ao invés de gerarmos amostras com todos os tamanhos de Δp.

Sabe-se que os valores de $\Delta\theta$ afetam a amplitude de Δp. Por esse motivo, pode-se limitar os valores de $\Delta\theta$ de forma que os tamanhos de Δp encaixem-se no intervalo desejado. Isso gera um problema, uma vez que precisamos decidir quais valores de $\Delta\theta$ devem ser utilizados.

Idealmente, gostaríamos de escolher com antecedência um intervalo de valores $\Delta\theta$ de deslocamentos angulares que resultassem em tamanhos conhecidos de Δp. Entretanto, este é exatamente o problema da cinemática inversa que estamos tentando resolver: calcular o $\Delta\theta$ necessário para gerar um determinado deslocamento Δp. Isso significa que é necessário escolher o intervalo de ângulos válidos a serem utilizados como $\Delta\theta$ empiricamente.

Ao escolher intervalos arbitrários empiricamente, nota-se que é muito difícil prever quais intervalos de ângulos de movimento gerarão somente amostras adequadas para o treinamento uma vez que um grande número de amostras sempre acaba contendo tamanhos de Δp maiores ou menores do que o desejado. Isso acontece porque Δp pode variar de uma maneira quase imprevisível se não levarmos em consideração a pose θ inicial.

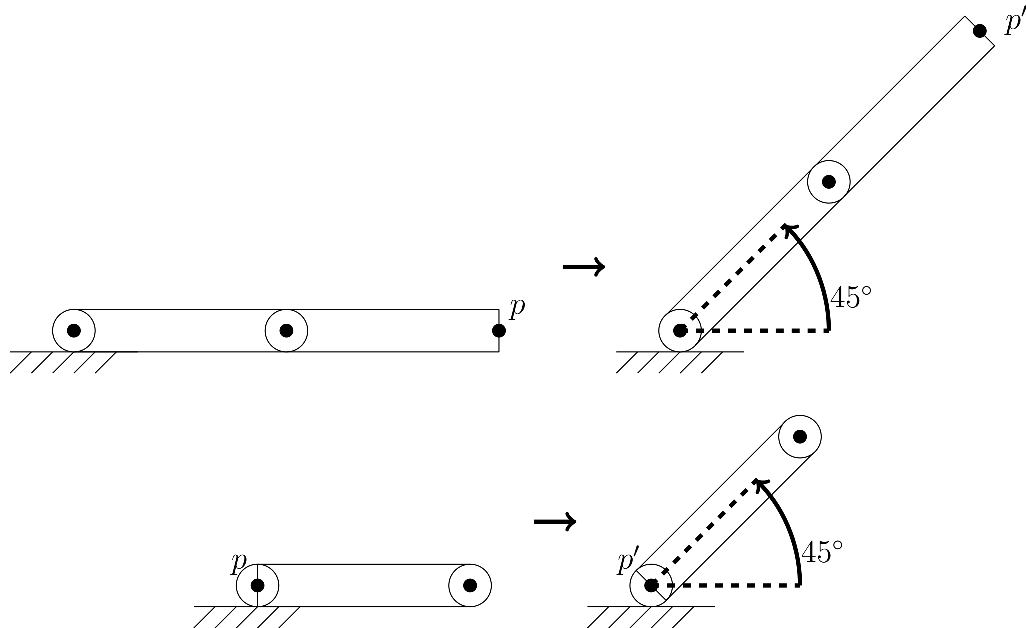
De maneira geral, pequenos valores de $\Delta\theta$ geram pequenos passos de Δp, enquanto valores maiores de $\Delta\theta$ geram movimentos maiores. Entretanto, isso nem sempre é verdade pois o mesmo conjunto $\Delta\theta$ de deslocamentos angulares pode gerar tamanhos de passos radicalmente diferentes, dependendo da pose inicial θ.

Como exemplo disso, tem-se dois casos de poses iniciais para um braço robótico de duas juntas (representados na Figura 4.4): no primeiro o braço está totalmente estendido, e no segundo seu “cotovelo” está contraído, posicionando o *end-effector* sobre a origem do braço. Pode-se observar que um mesmo movimento angular da primeira junta, quando aplicado a cada um dos dois casos, resultará em movimentos diferentes do *end-effector*. No primeiro caso, o movimento aplicado gerará um grande deslocamento do *end-effector*, enquanto, no segundo caso, o mesmo movimento gerará um deslocamento nulo.

Dessa forma, não existe uma forma prática de gerar apenas amostras que contenham deslocamentos Δp de tamanhos pré-determinados. De uma forma ou de outra, uma certa porcentagem das amostras geradas deverá ser descartada. Por isso, primeiro um movimento angular $\Delta\theta$ é gerado e aplicado para, em seguida, calcular-se o deslocamento Δp e analisar-se seu tamanho, certificando-se de que ele pertence ao intervalo decidido como válido.

Essa validação é feita no quarto passo da geração de dados, onde amostras maio-

Figura 4.4 – Um mesmo movimento de $45°$ na primeira junta gera resultados diferentes dependendo da pose inicial. No primeiro caso (acima), o braço está estendido. No segundo caso (abaixo), o "cotovelo" está contraído. p indica a posição inicial do *end-effector*, e p' sua posição após o movimento.



Fonte: Próprio autor.

res ou menores do que a amplitude decidida como válida para o treinamento da rede são descartadas.

Dadas essas ressalvas, em estágios iniciais do desenvolvimento da abordagem, os valores de $\Delta\theta$ gerados no segundo passo eram retirados de uma distribuição uniforme no intervalo decidido como válido. Esses intervalos são discutidos em profundidade no Capítulo 5. Entretanto, utilizar uma distribuição uniforme resultou em um número muito grande de amostras com tamanhos de Δp fora do intervalo válido e, portanto, inválidas.

A utilização de uma distribuição normal centrada em 0 gerou um maior número de amostras válidas, além de deixá-las mais semelhantes a situações do mundo real. Em aplicações reais, é muito mais comum ver movimentos com ângulos próximos de zero, por isso é importante treinar a rede neural com dados que representem essas situações. Para adquirir esses valores, geramos valores entre -1 e 1 e os escalamos para o intervalo desejado.

Sendo assim, a distribuição normal utilizada possui média 0 e desvio padrão $0{,}33$. A grande maioria dos valores contidos nessa distribuição se encontra entre -1 e 1, mas alguns valores acabam ficando fora do intervalo. Esses valores são descartados, resultando em uma distribuição normal na vizinhança de 0 truncada em -1 e 1. Os valores restantes são, então, escalados para os ângulos utilizados como máximo e mínimo para $\Delta\theta$. Para o caso de ângulos entre $-5°$ e $5°$, por exemplo, os valores gerados entre -1 e 1

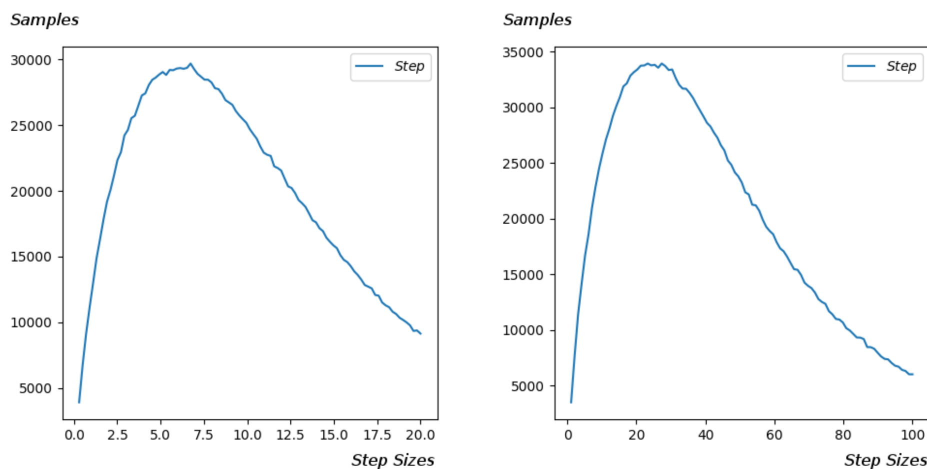
são escalados para o intervalo entre -5 e 5.

Para adquirir amostras com tamanhos de passos maiores para o *end-effector*, foram utilizados quatro intervalos distintos de $\Delta\theta$, com seus respectivos intervalos de tamanhos de passos resultantes. Os valores utilizados como máximo e mínimo para $\Delta\theta$ e o intervalo resultante de tamanhos de Δp são analisados em profundidade no Capítulo 5.

Para ilustrar a distribuição resultante das amostras em relação ao seus tamanhos de passos Δp, as Figuras 4.5 e 4.6 mostram histogramas das amostras geradas para um braço de duas juntas. No eixo horizontal são exibidos tamanhos de passos (tamanho de Δp), e no eixo vertical é exibido o número de amostras que resultam em deslocamentos de cada um desses tamanhos.

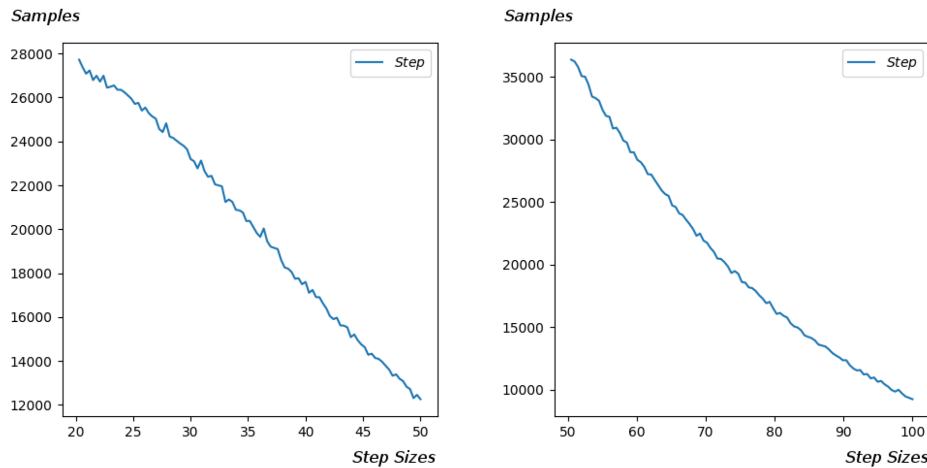
A forma do gráfico lembra uma distribuição normal inclinada, como é o caso dos gráficos da Figura 4.5. Isso acontece devido à influência de não-linearidades na cinemática do braço. Alguns dos intervalos que foram usados acabaram truncando a forma normal inclinada, mostrando apenas valores mais altos (a cauda da curva), resultando em uma curva descendente, como é o caso dos gráficos da Figura 4.6.

Figura 4.5 – Histogramas mostrando a relação entre o número de amostras e o tamanho dos passos para os intervalos de $0.1mm$ a $20.0mm$ e de $0.1mm$ a $100.0mm$, respectivamente.



Fonte: Próprio autor.

Figura 4.6 – Histogramas mostrando a relação entre o número de amostras e o tamanho dos passos para os intervalos de $20.0mm$ a $50.0mm$ e de $50.0mm$ a $100.0mm$, respectivamente.



Fonte: Próprio autor.

Isso também resultou em um número menor de amostras quando próximo do valor final do intervalo, e algumas vezes no começo do intervalo, se compararmos com o número de amostras no centro do intervalo.

O terceiro passo na geração dos dados é encontrar a nova posição p' do *end-effector* após aplicar o incremento dos valores de juntas na pose original do manipulador. Isso pode ser feito através do cálculo da cinemática direta.

Neste ponto, a configuração simples dos braços utilizados como caso de estudo neste trabalho mostram sua utilidade, uma vez que os braços planares utilizados como caso de estudo possuem uma solução facilmente calculável para o problema da cinemática direta.

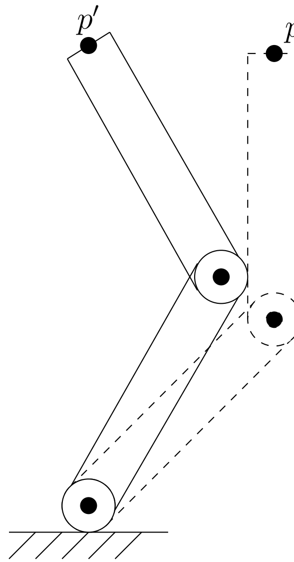
O processo utilizado para solucionar a cinemática direta e encontrar o ponto aqui chamado de p' é descrito na Seção 2.2.2, e uma representação é dada na Figura 4.7.

No quarto passo, o vetor Δp que representa a mudança na posição do *end-effector* deve ser calculado. Para isso, utilizamos a nova posição p' calculada no terceiro passo e a posição original p adquirida no primeiro passo. O cálculo de Δp é simples, e se dá através de $p' - p$.

Conforme explicado anteriormente, devido às complicações decorrentes do segundo passo, a amostra gerada precisa ser validada antes de ser inserida no *dataset*, e é neste último passo que isso ocorre. Para isso, analisamos o tamanho do vetor Δp para nos certificarmos de que a amostra se encaixa no intervalo delimitado, descartando amostras que resultam em movimentos fora do intervalo estabelecido como válido.

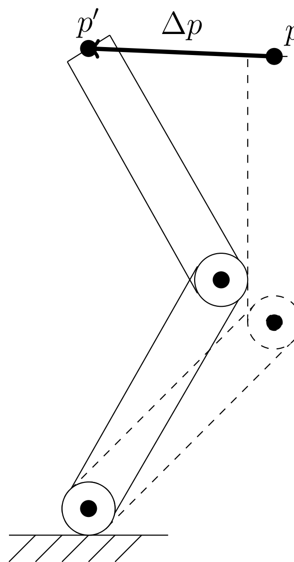
A Figura 4.8 mostra visualmente Δp relacionando p e p'.

Figura 4.7 – Passo 3: p' é a nova posição do *end-effector*, adquirida após o movimento ser aplicado.



Fonte: Próprio autor.

Figura 4.8 – Passo 4: O vetor Δp é calculado e seu tamanho é analisado para validação.



Fonte: Próprio autor.

A informação sobre Δp é armazenada utilizando os valores de Δx e Δy. Para um manipulador robótico que funciona no espaço 3D, ainda seria armazenado o valor de Δz.

Por fim, o segundo conjunto de entradas da rede neural é coletado, i.e., o vetor Δp representado através de Δx e Δy. Ainda nesse passo é coletada a saída da rede neural, ou seja, a variação nos valores de juntas $\Delta\theta$ gerado no segundo passo. Este último dado só pode ser coletado após a validação do tamanho de Δp, que é feita no quarto passo.

Colocados de maneira mais sucinta, os passos para a geração dos dados são:

1. Colocar o braço em uma pose aleatória (θ inicial) através da geração de ângulos aleatórios para cada junta e armazenar a posição p resultante do end-effector
2. Gerar uma pequena mudança aleatória nos valores de juntas ($\Delta\theta$)
3. Calcular a nova posição p' do end-effector através da cinemática direta
4. Calcular o vetor representando a mudança na posição do end-effector (Δx e Δy). Neste passo também são descartadas amostras cujo tamanho de Δp caem fora do intervalo desejado.

Os dados precisam ser alimentados à rede neural de maneira inversa, isto é, na ordem inversa que foram gerados. No processo de geração de dados utilizamos uma pose inicial θ e fazemos um pequeno movimento angular nas juntas ($\Delta\theta$) para só então calcularmos Δp. Ao invés disso, queremos informar à rede neural a pose inicial θ juntamente com o Δp desejado, para então recebermos como saída da rede neural o movimento angular $\Delta\theta$ necessário.

Os valores ainda precisam ser normalizados antes de serem alimentados à rede neural. Este processo é descrito na próxima seção.

4.3 NORMALIZAÇÃO DOS DADOS

Para alimentar os dados à rede neural, é preciso normalizar os valores contidos em cada amostra no intervalo de 0.0 a 1.0, para padronizar as entradas. Isso pode tornar o processo de treinamento mais rápido e reduzir as chances de o algoritmo ficar preso em um mínimo local.

Os valores dos ângulos iniciais θ das juntas são normalizados entre os valores máximos e mínimos válidos para uma junta, i.e., de $-\pi$ a π. Dessa forma, valores iguais a $-\pi$ são normalizados para 0.0, valores iguais a π são normalizados para 1.0, e valores intermediários são mapeados de acordo.

Os valores do segundo conjunto de entradas, a variação Δp desejada na posição do *end-effector*, são normalizados no intervalo de movimentos válidos decididos no quarto passo do respectivo processo de geração de dados. Esses intervalos são examinados em mais detalhes no Capítulo 5.

Neste ponto do processo, consideremos como exemplo o *dataset* que contém amostras cujos tamanhos de Δp se encontram no intervalo entre $20mm$ e $50mm$. Aqui, a normalização significa que amostras com tamanhos de Δp iguais a $20mm$ serão mapeadas para

0.0, e amostras com tamanhos de Δp iguais a $50mm$ serão mapeadas para 1.0, enquanto os tamanhos intermediários serão mapeados entre 0.0 e 1.0, de acordo com seus valores.

O vetor Δp que descreve o movimento resultante do *end-effector* pode apontar para qualquer direção, então os valores de Δx e Δy podem variar consideravelmente. Ainda assim, é possível prever os valores máximos e mínimos de Δx e Δy.

Considerando que os vetores são validados de acordo com seu tamanho, sabemos o tamanho máximo e mínimo possível do vetor Δp de uma dada amostra. Para esse vetor, a variação máxima no eixo x acontecerá quando sua componente y não muda, i.e., quando o vetor de tamanho máximo está exatamente sobre o eixo x. Dessa forma, se o vetor Δp possui tamanho máximo igual a $50.0mm$, isso significa que Δx pode ir até 50, e isso acontece quando Δy é igual a 0. Isso pode acontecer em ambas as direções do eixo x, então pode-se prever que os valores de Δx estarão sempre entre -50 e 50. Através do mesmo raciocínio, concluímos que o mesmo se aplica para Δy.

Por esse motivo, os valores de Δx e Δy são simplesmente normalizados entre os valores máximo e mínimo contidos nas amostras, valor esse que é truncado no quarto passo da geração dos dados.

Os valores das saídas da rede neural, isto é, a mudança incremental $\Delta\theta$ nos ângulos das juntas que irão causar o movimento desejado no *end-effector*, são normalizados nos intervalos originalmente utilizados para gerar os movimentos aleatórios no segundo passo da geração de dados. Os valores utilizados como máximos e mínimos para os casos estudados são mostrados no Capítulo 5.

Como os dados são todos normalizados para valores entre 0.0 e 1.0 antes de serem utilizados no treinamento da rede neural, ela acaba desconhecendo a natureza original desses valores. Por esse motivo, sua saída, que possuirá valores entre 0.0 e 1.0, deverá sofrer o processo oposto da normalização antes de ser aplicada no manipulador. Ou seja, os valores devem ser reescalados para seus intervalos originais.

Na próxima seção são apresentados detalhes técnicos a respeito da implementação da rede neural e a respeito dos valores utilizados como parâmetros de treinamento.

4.4 ARQUITETURA DA REDE NEURAL

Para o processo de treinamento e implantação da rede neural, utilizou-se o *framework* Keras (CHOLLET et al., 2015), que funciona através da linguagem Python. O *framework* permite a fácil criação de modelos de redes neurais e fornece ferramentas que ajudam o desenvolvedor a lidar com os parâmetros de treinamento.

A rede neural utilizada neste trabalho é uma simples rede *feed-forward* multi-camada totalmente conectada. Conforme explicado mais a frente, no Capítulo 5, utilizaram-se dois braços robóticos, um com 2 DOF e outro com 3 DOF. A rede utilizada para o manipula-

dor de duas juntas possui quatro neurônios de entrada, enquanto a rede utilizada para o manipulador de três juntas possui cinco neurônios, como é exibido na Tabela 4.1.

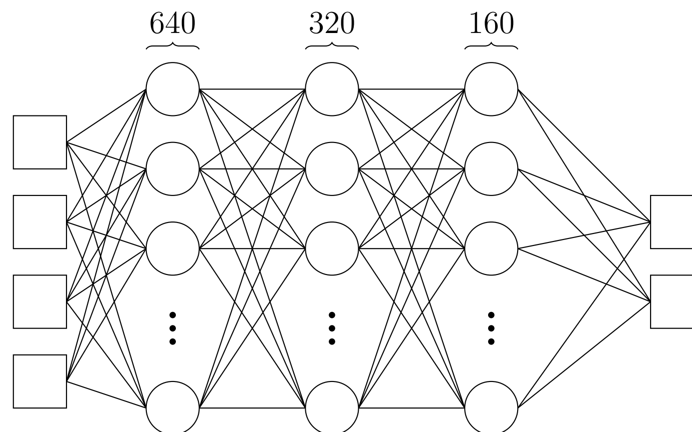
Tabela 4.1 – Entradas e saídas para os braços planares de duas e três juntas.

Entrada/Saída	Braço planar de 2 juntas	Braço planar de 3 juntas
Entradas	θ_1, θ_2, Δx, Δy	θ_1, θ_2, θ_3, Δx, Δy
Saídas	$\Delta\theta_1$, $\Delta\theta_2$	$\Delta\theta_1$, $\Delta\theta_2$, $\Delta\theta_3$

Fonte: Próprio autor.

A rede neural usada possui três camadas escondidas de neurônios, sendo que a primeira camada possui 640 neurônios, a segunda possui 320 e, a terceira, 160 neurônios. Vários experimentos foram feitos e essa arquitetura de rede acabou sendo a que obteve os melhores resultados. A Figura 4.9 ilustra a quantidade de neurônios por camadas. Dadas as conexões entre os neurônios e os pesos de cada conexão, a rede totaliza 260.803 parâmetros treináveis.

Figura 4.9 – Visualização da quantidade de neurônios por camada da rede neural.



Fonte: Próprio autor.

Segundo Srivastava et al. (2014), *overfitting* é um sério problema que ocorre em redes neurais grandes. *Overfitting* se refere ao fato de que redes neurais podem "decorar" os conjuntos de entrada e saída apresentados a ela durante o treinamento, sem necessariamente aprender uma solução genérica para o problema. Isto é, a rede oferecerá respostas precisas para os dados utilizados durante o treinamento, mas não necessariamente uma boa resposta para dados novos, nunca vistos pela rede neural. O dicionário Oxford define *overfitting* como:

> "A produção de uma análise que corresponde muito aproximadamente ou exatamente a um conjunto particular de dados, e pode, portanto, falhar em encaixar dados adicionais ou prever futuras observações de maneira confiável."

Por esse motivo, na rede neural utilizada nesse trabalho foi aplicada a técnica de *dropout* na saída da segunda e da terceira camada escondida. *Dropout* é uma técnica que apaga neurônios e suas conexões da rede neural aleatoriamente durante o treinamento. Isso previne que neurônios se co-adaptem em excesso (SRIVASTAVA et al., 2014).

Os neurônios das camadas escondidas utilizam uma unidade linear retificada (ReLU) como função de ativação, enquanto os neurônios da camada de saída utilizam uma função sigmoide como função de ativação.

Cada modelo de rede neural treinado durante este trabalho utilizou erro quadrático médio (MSE) como métrica de *loss* e uma taxa de aprendizagem de 0.002, além de utilizar o otimizador Adam (KINGMA; BA, 2014) ao invés do gradiente descendente estocástico, que é o algoritmo clássico utilizado para atualizar os pesos da rede neural durante o processo de treinamento. De acordo com o autor, o método Adam – cujo nome é derivado do termo *adaptative moment estimation* – computa taxas de aprendizagem adaptativas individuais para diferentes parâmetros a partir de estimativas do primeiro e segundo momentos dos gradientes, diferentemente do método do gradiente descendente estocástico, que mantém uma única taxa de aprendizagem para todas as atualizações de pesos e que não muda durante o treinamento.

A configuração de todos esses parâmetros é feita de maneira simples através do *framework* Keras (CHOLLET et al., 2015). A utilização do *framework* Keras fica clara no código-fonte exibido no Anexo A.

No total, foram utilizados 2 milhões de amostras para cada um dos intervalos apresentados no Capítulo 5, para cada um dos dois manipuladores, totalizando oito *datasets* e 16 milhões de amostras. O tamanho de *batch* utilizado para o treinamento foi de 2000, e cada modelo foi treinado por 100 épocas. O número total de modelos de redes neurais foi oito – quatro para cada manipulador – de acordo com os devidos intervalos.

4.5 SUMÁRIO DO CAPÍTULO

Neste capítulo apresentou-se a abordagem de maneira detalhada. A geração das amostras para o treinamento da rede neural se dá em quatro passos e, apesar das nuances que aumentam a complexidade do processo, os passos mantém-se relativamente simples.

Após a geração das amostras, elas são normalizadas para serem utilizadas no treinamento da rede neural. Consequentemente, após treinada, a rede entregará na saída valores normalizados que deverão ser reescalados para os intervalos originais antes de serem utilizados nos manipuladores.

No próximo capítulo será apresentado o caso de estudo utilizado para a validação da abordagem.

5 CASO DE ESTUDO

Para validação da metodologia, foram utilizados dois braços planares: um com duas juntas de revolução e outro com três juntas de revolução. Ambos esses braços possuem solução fechada bem conhecida para a cinemática inversa, mas, conforme mencionado em trabalhos relacionados citados nesse texto, essas duas configurações de braços robóticos são excelentes plataformas de testes para abordagens que visam resolver o problema da cinemática inversa. Os braços planares de 2-DOF e 3-DOF foram escolhidos por clareza e por se encaixarem bem na abordagem desenvolvida neste trabalho.

Para esse fim, como prova de conceito para o teste da metodologia apresentada neste trabalho, ambos os braços planares foram simulados em um ambiente 2D criado através da utilização da biblioteca gráfica PyGame (SHINNERS, 2011). Nessa simulação, foram utilizados dois braços com 600 unidades de comprimento, representando braços robóticos de $60cm$. Este tamanho foi escolhido por ser o tamanho aproximado de muitos manipuladores, incluindo robôs presentes no laboratório de robótica no qual o trabalho foi desenvolvido. O braço de duas juntas possui dois segmentos de $30cm$ cada, enquanto o braço de três juntas possui três segmentos de $20cm$ cada.

Um possível lado negativo deste tamanho é que as duas juntas do manipulador de 2-DOF fornecem uma resolução relativamente pobre uma vez que movimentos pequenos e precisos do *end-effector* exigem tamanhos extremamente pequenos de ângulos para mover os segmentos relativamente longos ($30cm$) do braço. Apesar disso, a decisão de utilizar esse tamanho de braço foi baseada na intensão de garantir mais confiabilidade à simulação, uma vez que o tamanho é baseado em braços robóticos reais.

A utilização do Jacobiano para mover os braços robóticos é importante para a comparação de sua performance em relação à da rede neural mas, como visto anteriormente, este processo pode ser trabalhoso. A utilização do braço robótico de dois segmentos e duas juntas em um espaço 2D resulta em uma matriz Jacobiana quadrada, com duas linhas e duas colunas. Essa matriz pode ser calculada e invertida facilmente e, por essa razão, o braço planar de duas juntas foi o escolhido.

O braço de três juntas não gera uma matriz quadrada, e portanto não é possível utilizar a inversa do Jacobiano. Apesar disso, conforme comentado anteriormente, ainda pode-se utilizar a pseudo-inversa do Jacobiano para a solução da cinemática de velocidade. Desta forma, o braço de três juntas adiciona uma complexidade a mais na validação, sem prejudicar os testes.

Quatro intervalos distintos foram utilizados para a geração de dados e, consequentemente, quatro redes neurais foram treinadas para cada braço robótico. O objetivo foi gerar redes neurais treinadas especificamente para diferentes intervalos de passos. Esses intervalos são mostrados na Tabela 5.1.

Tabela 5.1 – Valores utilizados para a geração de amostras.

$\Delta\theta$ mínimo	$\Delta\theta$ máximo	Tamanho mínimo de Δp	Tamanho máximo de Δp
$-5°$	$5°$	$0.1mm$	$20.0mm$
$-15°$	$15°$	$20.0mm$	$50.0mm$
$-20°$	$20°$	$50.0mm$	$100.0mm$
$-20°$	$20°$	$0.1mm$	$100.0mm$

Fonte: Próprio autor.

Conforme dito anteriormente, o método Jacobiano é muito preciso para passos pequenos. Por esse motivo, uma rede foi treinada para passos de tamanhos entre $0.1mm$ e $20.0mm$. Além disso, o objetivo era gerar redes neurais capazes de ter uma performance melhor que a do Jacobiano para intervalos maiores, onde o Jacobiano não é tão preciso. Por esse motivo, o segundo e o terceiro intervalos mencionados na Tabela 5.1 foram utilizados. Por fim, um intervalo que cobre todos os outros intervalos foi utilizado para o treinamento de uma última rede neural.

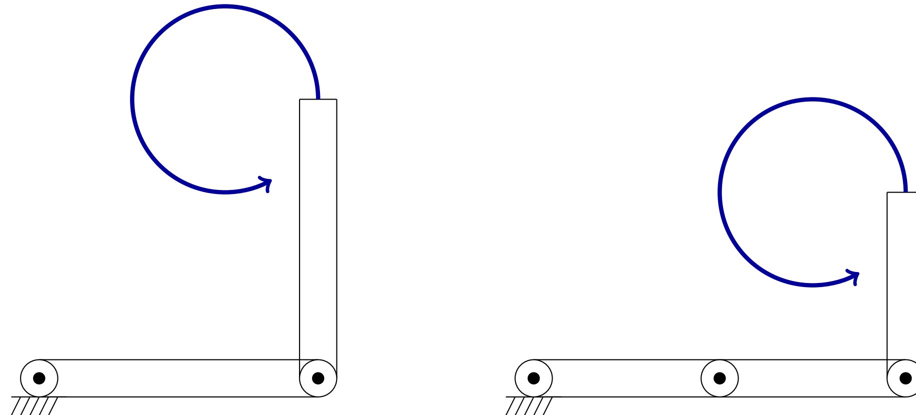
A Tabela 5.1 ainda relaciona os intervalos de tamanhos de Δp com os intervalos de ângulos de $\Delta\theta$ utilizados para a geração dos movimentos.

Após o treinamento da rede neural, projetou-se uma tarefa para comparar seu desempenho em relação ao desempenho do método do Jacobiano inverso. Como não existe um *benchmark* específico para o teste deste tipo de metodologia, a tarefa utilizada foi baseada em tarefas semelhantes observadas nos trabalhos relacionadas. A tarefa seria controlar os manipuladores de duas e três juntas de maneira que o *end-effector* seguisse uma trajetória circular de raio $100mm$ com diferentes tamanhos de incrementos. A trajetória para o braço de duas juntas possui seu centro nas coordenadas $(200, 300)$, enquanto o centro da trajetória utilizada para o braço de três juntas é nas coordenadas $(300, 200)$.

Essa diferença no posicionamento da trajetória desejada foi criada com o objetivo de podermos colocar o *end-effector* do manipulador facilmente sobre o começo da trajetória para iniciarmos os testes. Para casos em que o braço começava fora da trajetória, o primeiro movimento do braço causava uma distorção nos dados uma vez que esse primeiro passo acabava tendo um tamanho diferente do que os pontos pertencentes à trajetória. A Figura 5.1 mostra as poses inicias para os manipuladores de duas e três juntas, respectivamente, e também suas posições relativas às suas trajetórias. Ambos os braços seguem suas respectivas trajetórias, representadas em azul na Figura 5.1, em um movimento anti-horário.

A tarefa é repetida várias vezes, cada vez utilizando tamanhos diferentes de passos. Em cada rodada de testes, a tarefa é executada por um total de 1000 pontos uniformemente distribuídos na trajetória, utilizando um ponto após o outro como posição alvo do *end-effector*. A cada rodada de testes, a distância entre cada ponto, isto é, o tamanho do passo que o *end-effector* deve dar, aumenta incrementalmente.

Figura 5.1 – Poses iniciais dos manipuladores. Linhas pretas representam os segmentos do braço; o círculo azul representa a trajetória.



Fonte: Próprio autor.

Na primeira rodada de testes, o tamanho do passo é o mínimo. Depois que o manipulador percorre todos os 1000 pontos dados na trajetória, essa rodada é encerrada. Então, o erro quadrático médio (MSE) é calculado para ambos os métodos de controle, e a tarefa é reiniciada para a próxima rodada, com os pontos da trajetória espaçados em distâncias incrementalmente maiores, e o processo é repetido.

Utilizando como exemplo a rede neural treinada nos intervalos entre $0.1mm$ e $20mm$, a primeira rodada de 1000 passos será feita utilizando como alvos pontos uniformemente espaçados sobre a trajetória com uma distância de $0.1mm$ entre cada um. Após essa rodada, uma nova rodada será feita utilizando pontos com uma distância de $0.2mm$ entre eles, até que a distância de $20mm$ seja alcançada. Os incrementos se dão de $0.1mm$ em $0.1mm$.

Conforme dito anteriormente no texto, no total, foram treinadas e comparadas a precisão do controle que utiliza a rede neural com o controle do método do Jacobiano inverso em quatro intervalos distintos, sendo eles: (1) de $0.01mm$ a $20mm$, (2) de $20mm$ a $50mm$, (3) de $50mm$ a $100mm$, e uma trajetória que engloba todo o intervalo (4) de $0.1mm$ a $100mm$. Para cada um dos intervalos foi utilizada uma rede neural com pesos treinados especificamente para aquele intervalo.

5.1 SUMÁRIO DO CAPÍTULO

A validação da metodologia foi feita através de braços simulados em um ambiente 2D. Os braços de 2-DOF e 3-DOF possuem configurações de solução simples que possuem uma fácil comparação entre o desempenho da rede neural e do método Jacobiano.

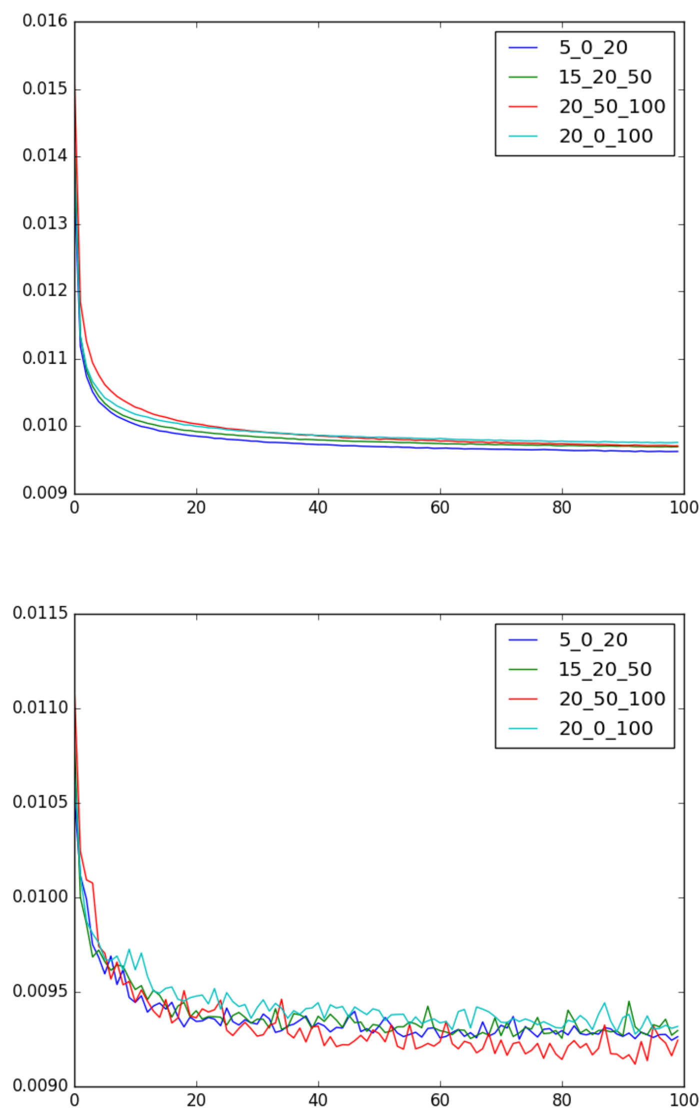
Quatro intervalos distintos de dados foram utilizados para o treinamento de redes neurais que controlam ambos os braços robóticos, totalizando oito redes neurais treinadas.

Após o treinamento, uma tarefa foi proposta para comparar os desempenhos da rede e do Jacobiano. Os resultados da comparação são apresentados no próximo capítulo.

6 RESULTADOS

Os modelos das quatro redes neurais criadas, uma para cada intervalo de dados, foram treinados resultando em gráficos de natureza similar aos exibidos na Figura 6.1[1] que descreve o valor de *training loss* e *validation loss* das redes treinadas para o braço de três juntas.

Figura 6.1 – *Traning loss* e *validation loss* das redes neurais treinadas para o braço de três juntas, respectivamente.

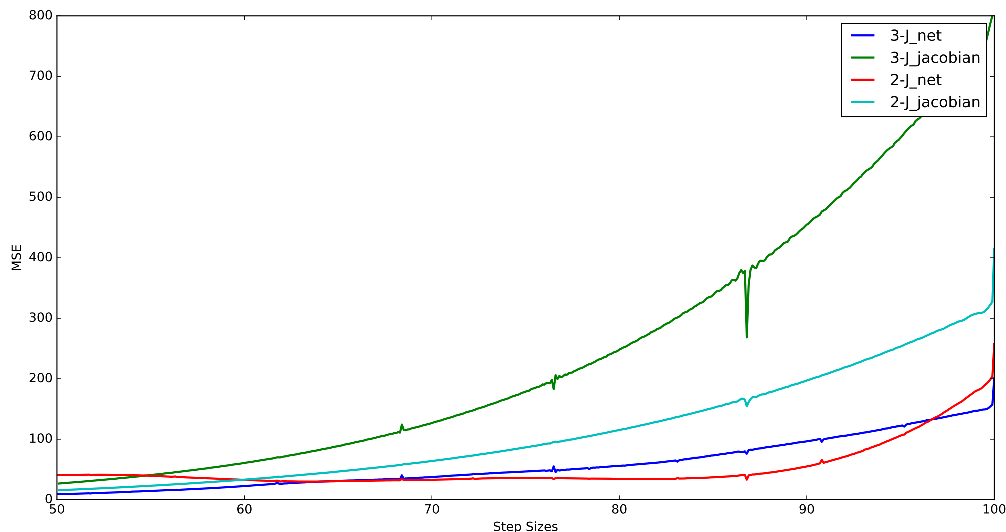


Fonte: Montenegro et al. (2018)

[1] As curvas estão legendadas no formato [*limite de variação de ângulo*]_[*tamanho mínimo do passo*]_[*tamanho máximo do passo*], referindo-se aos *datasets* utilizados para o treinamento.

Em geral, os testes mostraram que a rede neural é muito mais estável do que o método que utiliza o Jacobiano inverso para o intervalo no qual a rede foi treinada. Esse comportamento pode ser visto claramente na Figura 6.2, que mostra a comparação entre a precisão de ambos os braços na trajetória testada quando controlados pelo método da (pseudo)inversa do Jacobiano e pela rede neural no intervalo entre $50mm$ e $100mm$.

Figura 6.2 – Braços de duas juntas (2-J) e três juntas (3-J) executando a tarefa no intervalo entre $50mm$ e $100mm$.



Fonte: Montenegro et al. (2018)

O eixo vertical do gráfico da Figura 6.2 mostra o MSE, enquanto o eixo horizontal mostra os tamanhos (em mm) de passos utilizados na trajetória, relacionando o desempenho de cada método ao controlar os braços testados. A curva em verde mostra o desempenho do Jacobiano ao controlar o braço de três juntas, e a curva em azul, o desempenho da rede neural ao controlar o mesmo braço. As curvas em vermelho e ciano mostram os desempenhos da rede neural e do Jacobiano, respectivamente, ao controlar o braço de duas juntas.

Como esperado, a rede neural tem um desempenho pior do que o Jacobiano inverso para tamanhos de passos pequenos, como mostrado no inicio do intervalo na Figura 6.2. Note que a rede neural é pior do que o Jacobiano em intervalos pequenos tanto ao controlar o braço de duas juntas (curvas em vermelho e ciano) quanto o braço de três juntas (curvas em azul e verde).

Por outro lado, a rede neural tem um desempenho muito melhor para tamanhos de passos maiores (curvas em azul e vermelho). Mesmo quando a rede neural tem sua pior performance, perto do fim do intervalo, ela ainda tem MSE muito menor que do Jacobiano inverso. Observe, no intervalo de passos com tamanho entre 90 e 100, como o desem-

penho da rede neural ao controlar o braço de duas juntas (curva em vermelho), mesmo piorando, ainda continua muito mais preciso do que o Jacobiano controlando o mesmo braço (curva em ciano).

Conforme os tamanhos dos passos aumentam, o MSE do Jacobiano aumenta exponencialmente, enquanto o MSE da rede neural permanece quase constante, somente aumentando um pouco mais rapidamente quando próximo ao final do intervalo treinado. Esse desempenho pior da rede neural aparenta não depender da complexidade do problema ou do tamanho maior de passos, mas sim do fato de que existem menos amostras com Δp deste tamanho, conforme ilustrado pela Figura 4.6.

Como um teste final, comparou-se o comportamento de ambos os métodos quando próximo a uma singularidade. Para este teste, colocou-se o manipulador de duas juntas em uma posição inicial com o braço totalmente estendido para a direita, e criou-se uma trajetória na qual o *end-effector* faz um movimento horizontal para a esquerda, até a origem do sistema, e então um movimento para cima, criando uma forma de "L".

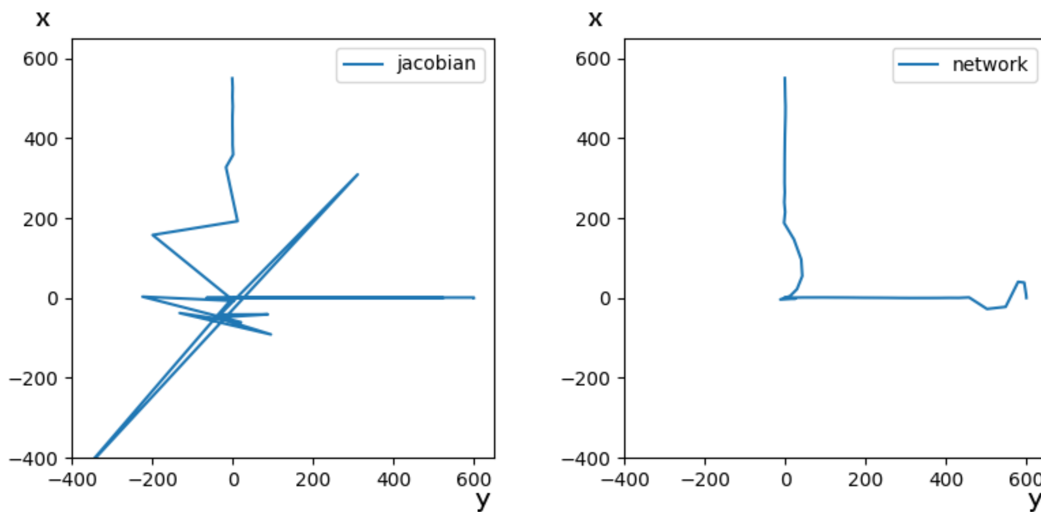
A origem do sistema também é a posição da primeira junta e, por ambos os segmentos do braço terem o mesmo tamanho, isso configura uma singularidade do sistema.

Como pode ser visto na Figura 6.3, que mostra a trajetória descrita pelo *end-effector* ao tentar seguir a trajetória proposta, a rede neural é muito mais estável do que o Jacobiano quando próximo a singularidades. A trajetória descrita pelo *end-effector* quando controlado pelo Jacobiano comete erros agudos quando está próximo à origem, enquanto a rede neural controla o manipulador de uma forma suave por toda a trajetória.

Isso acontece porque, quando próximo à singularidade – a origem do sistema – o método Jacobiano acaba envolvendo uma divisão por valores que tendem a zero, o que gera movimentos de juntas que tendem ao infinito. Por esse motivo, pode-se observar que o *end-effector* acaba realizando movimentos agudos que não condizem com a trajetória.

A rede neural modela o problema internamente baseando-se apenas nas amostras que foram apresentadas a ela durante o treinamento para resolver o problema, ao invés de fazer o cálculo de uma maneira puramente "matemática", desvinculada da experiência prática. Isso significa que, quando modelado pela rede neural, o problema não necessariamente envolve divisões por zero, contornando o problema causado pela singularidade no método Jacobiano.

Figura 6.3 – Comparação do desempenho do método do Jacobiano inverso e da rede neural quando próximo a uma singularidade.



Fonte: Montenegro et al. (2018)

6.1 SUMÁRIO DO CAPÍTULO

Apesar de possuir resultados não tão precisos quanto os do Jacobiano para intervalos pequenos, as redes neurais treinadas da forma apresentada neste trabalho apresentam resultados satisfatórios. A precisão da rede é melhor do que a do Jacobiano para intervalos maiores e, mesmo em intervalos menores, seu desempenho é mais previsível. Isso significa que o desempenho da rede neural nos intervalos onde ela foi treinada é quase constante, independentemente do tamanho de Δp. Diferentemente do método Jacobiano que, por regra, possui uma pior precisão quanto maior for o tamanho de Δp.

O desempenho pior da rede neural quando próximo do limite máximo de tamanho de Δp aparenta não depender do tamanho de Δp em si, mas sim do número menor de amostras que contém movimentos que resultam nestes tamanhos.

7 CONCLUSÃO E TRABALHOS FUTUROS

Nessa dissertação, apresentou-se um método para o treinamento de uma rede neural para computar a cinemática diferencial inversa de manipuladores robóticos. Para validar os resultados, comparou-se a precisão da rede neural à do método Jacobiano. Utilizaram-se dois braços planares com duas e três juntas, e uma trajetória circular foi proposta como tarefa para avaliar o desempenho de ambas as abordagens.

Para cada braço robótico, utilizaram-se quatro intervalos diferentes de interesse, resultando em quatro redes neurais diferentes treinadas com conjuntos únicos de dados. No total, oito modelos de redes neurais foram treinados.

Baseando-se nos resultados observados, nota-se que a utilização da abordagem proposta nesse trabalho pode ter vantagens quando aplicada de uma maneira híbrida. Em uma abordagem híbrida, a rede neural poderia ser utilizada para mover o robô incrementalmente em passos maiores, enquanto o *end-effector* está longe de sua posição objetivo. Conforme o *end-effector* se aproxima do objetivo, o sistema poderia automaticamente alterar o modo de controle para dar preferência ao método Jacobiano inverso. Para isso a rede neural precisa conhecer a posição atual do *end-effector*, que pode ser facilmente adquirida através da cinemática direta. O método Jacobiano é muito preciso para passos incrementais de tamanhos pequenos, e pode ser utilizado para um ajuste fino, até que a posição destino seja alcançada.

Desta maneira, o braço chegaria até a posição alvo em menos passos, permitindo um menor número de iterações. Através de passos iniciais maiores, o tempo total para a execução da tarefa seria potencialmente diminuído, dependendo da aceleração do *hardware* e da complexidade do robô. O uso dessa abordagem torna possível a utilização de uma GPU para rodar a rede neural, possivelmente diminuindo o custo computacional para a CPU.

Além disso, passos maiores implicam uma menor necessidade de largura de banda para o laço de controle no sistema que comunica-se com os atuadores.

Para trabalhos futuros, o projeto feito pela equipe que desenvolveu a abordagem aqui apresentada ainda inclui a aplicação desta técnica em robôs reais. Uma primeira versão da técnica já foi validada em uma versão simulada do robô Thormang (ROBOTIS, 2018), que possui um braço com sete juntas e 7-DOF e está em processo de implantação para testes em uma versão do robô no mundo real. A mesma abordagem já está em desenvolvimento em uma versão real do robô Thormang em parceria com o *Educational Robotics Center*, do *Department of Electrical Engineering da National Taiwan Normal University*. Além disso, sugere-se que seja desenvolvido como trabalho futuro a avaliação aprofundada do custo computacional da abordagem.

REFERÊNCIAS BIBLIOGRÁFICAS

AGGARWAL, L.; AGGARWAL, K.; URBANIC, R. J. Use of artificial neural networks for the development of an inverse kinematic solution and visual identification of singularity zone (s). **Procedia Cirp**, Elsevier, v. 17, p. 812–817, 2014.

ALMUSAWI, A. R.; DÜLGER, L. C.; KAPUCU, S. A new artificial neural network approach in solving inverse kinematics of robotic arm (denso vp6242). **Computational intelligence and neuroscience**, Hindawi, v. 2016, 2016.

ASIMOV, I. 4 the word i invented. **Counting the Eons**, 1983.

ASIMOV, I.; SILVERBERG, R. The robot chronicles. **Gold–The Final Science Fiction Collection. HarperPrism**, 1995.

BERRESFORD, G. C.; ROCKETT, A. M. **Brief applied calculus**. [S.l.]: Nelson Education, 2015.

BINGUL, Z.; ERTUNC, H.; OYSU, C. Comparison of inverse kinematics solutions using neural network for 6r robot manipulator with offset. In: IEEE. **Computational Intelligence Methods and Applications, 2005 ICSC Congress on**. [S.l.], 2005. p. 5–pp.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

CRAIG, J. J. **Introduction to robotics: mechanics and control**. [S.l.]: Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2005. v. 3.

DENAVIT, J.; HARTENBERG, R. S. A kinematic notation for lower-pair mechanisms based on matrices. **ASME Journal of Applied Mechanics**, v. 23, p. 215–221, 1955.

DUKA, A.-V. Neural network based inverse kinematics solution for trajectory tracking of a robotic arm. **Procedia Technology**, Elsevier, v. 12, p. 20–27, 2014.

HASAN, A. T. et al. Artificial neural network-based kinematics jacobian solution for serial manipulator passing through singular configurations. **Advances in Engineering Software**, Elsevier, v. 41, n. 2, p. 359–367, 2010.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

KÖKER, R. A genetic algorithm approach to a neural-network-based inverse kinematics solution of robotic manipulators based on error minimization. **Information Sciences**, Elsevier, v. 222, p. 528–543, 2013.

KOZA, J. R. et al. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In: **Artificial Intelligence in Design96**. [S.l.]: Springer, 1996. p. 151–170.

LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **nature**, Nature Publishing Group, v. 521, n. 7553, p. 436, 2015.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MONTENEGRO, F. J. C. **EzGame**. 2018. <https://github.com/SplinterDev/ezgame>.

____. **Point2D**. 2018. <https://github.com/SplinterDev/point2d>.

MONTENEGRO, F. J. C. et al. Neural network as an alternative to the jacobian for iterative solution to inverse kinematics. In: IEEE. **2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)**. [S.l.], 2018. p. 333–338.

NEEDHAM, J.; RONAN, C. A. **The shorter science and civilisation in China**. [S.l.]: Cambridge University Press, 1995. v. 5.

NOF, S. Y. **Handbook of industrial robotics**. [S.l.]: John Wiley & Sons, 1999. v. 1.

ORT, T. **Art and Life in Modernist Prague: Karel Čapek and His Generation, 1911-1938**. [S.l.]: Springer, 2013.

OYAMA, E. et al. A modular neural network architecture for inverse kinematics model learning. **Neurocomputing**, Elsevier, v. 38, p. 797–805, 2001.

ROBOTIS. **THORMANG3 e-Manual**. 2018. Disponível em: <http://emanual.robotis.com/docs/en/platform/thormang3/introduction/>.

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence: a modern approach**. [S.l.]: Malaysia; Pearson Education Limited„ 2016.

SHARKEY, N. A programmable robot from 60 ad, 2611. **New Scientist**, 2007.

SHINNERS, P. **PyGame**. 2011. <http://pygame.org/>.

SICILIANO, B. et al. **Robotics: modelling, planning and control**. [S.l.]: Springer Science & Business Media, 2010.

SPONG, M. W. et al. **Robot modeling and control**. [S.l.]: Wiley New York, 2006. v. 3.

SRIVASTAVA, N. et al. Dropout: a simple way to prevent neural networks from overfitting. **The Journal of Machine Learning Research**, JMLR. org, v. 15, n. 1, p. 1929–1958, 2014.

TCH, A. **The mostly complete chart of Neural Networks, explained**. 2017. Acessado em: 15/02/2019. Disponível em: <https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464>.

ANEXO A – CÓDIGO-FONTE DESENVOLVIDO

O código-fonte foi desenvolvido em Python e dividido em vários arquivos. Um arquivo principal chamado *application.py* determina o modo de funcionamento da simulação a partir de quatro possibilidades:

- Geração de amostras
- Execução da tarefa através do método Jacobiano
- Execução da tarefa através da rede neural
- Visualização de poses aleatórias

Os arquivos estão separados da seguinte maneira:

application.py
: Determina o modo de funcionamento da simulação e a executa.

arm.py
: Define uma classe que representa o braço robótico, com métodos para calcular a cinemática direta do braço, medir o erro em relação a um dado alvo, entre outras funcionalidades.

helper.py
: Contém funções auxiliares.

jacobian.py
: Define a classe *JacobianController* que é utilizada para calcular o movimento a ser feito pelo braço a partir de um dado alvo utilizando o método Jacobiano.

logger.py
: Formata os dados analisados e os salva em arquivos.

net.py
: Define a classe *Net* que é utilizada para calcular o movimento a ser feito pelo braço a partir de um dado alvo utilizando um modelo de rede neural treinado.

train_net.py
: Treina a rede neural utilizando o *framework* Keras.

trajectory.py
: Gera os pontos da trajetória e torna possível que os controladores dos braços robóticos consultem alvos de maneira iterativa.

Para o desenvolvimento da simulação foram criadas duas bibliotecas. A biblioteca *Point2D* facilita a criação e manipulação geométrica de vetores 2D no plano (MONTENEGRO, 2018b). A biblioteca *EzGame* (MONTENEGRO, 2018a) cria uma camada sobre a bilbioteca gráfica PyGame e facilita a exibição de pontos e retas no plano. O código anexado é o utilizado para o braço de duas juntas, mas a adaptação para o braço de três juntas é trivial. O código também está disponível online[1].

A.1 – *APPLICATION.PY*

```
from ezgame import Ezgame
from point2 import Point2
import numpy as np
import csv

from arm import *
from trajectory import Trajectory
from jacobian import JacobianControler
from net import Net
from helper import *
from logger import Logger

FPS = 100000


# APPLICATION MODES
JACOBIAN         = 0
RANDOM_POSE      = 1
GENERATE_SAMPLES = 2
TEST_NETWORK     = 3
MODELID = "20180714_154336"

CURRENT_MODE = GENERATE_SAMPLES

class Application:
    def __init__(self):
        self.ez = Ezgame(700, 700)
        self.ez.fps = FPS
        self.loop = CURRENT_MODE
```

[1] https://github.com/SplinterFM/JacobianNNPlanar

```
        self.arm = Arm()
        self.init()


    def init(self):
        if self.loop == JACOBIAN:
            self.ez.init(self.jacobian_loop)
            # needs a trajectory
            self.traj_step = TRAJ_STEP_MIN
            self.trajectory = Trajectory(
                step_length=self.traj_step,
                restart=self.reset)
            self.jacobian  = JacobianControler(self.arm)
            self.errors = []
            self.plot_errors = []
            self.jacobian_logger = Logger("jacobian_control")
            # self.ez.gui = False

        elif self.loop == RANDOM_POSE:
            self.ez.init(self.random_pose_loop)
            self.arm2 = Arm()
            self.ez.fps = 2

        elif self.loop == GENERATE_SAMPLES:
            self.ez.init(self.generate_samples_loop)
            self.arm2 = Arm()
            self.ez.gui = False

            # the file has in its name the range of delta theta in degrees
            # and the maximum step valid
            filename = "dataset_{}_{}-{}.csv".format(DT_VAL, MIN_STEP, MAX_STEP)
            csvfile = open(filename, 'a')
            self.dataset_writer = csv.writer(csvfile)
            self.sample_counter = 0
            self.invalid_counter = 0

        elif self.loop == TEST_NETWORK:
            self.ez.init(self.test_net_loop)
            # needs a trajectory
            self.traj_step = TRAJ_STEP_MIN
            self.trajectory = Trajectory(
```

```
                step_length=self.traj_step,
                restart=self.reset)
            modelid = MODELID
            self.net = Net(modelid, self.arm)
            self.errors = []
            self.net_logger = Logger("net_control{}".format(modelid))


    def test_net_loop(self):
        self.draw_arm(self.arm)
        self.draw_trajectory()

        # get current position
        t1  = self.arm.t1
        t2  = self.arm.t2
        ee1 = self.arm.endeffector()

        self.current_point = self.trajectory.current()
        self.net.control(self.current_point)
        self.errors.append(self.arm.error(self.current_point))
        self.draw_info()

        self.current_point = self.trajectory.next()

        self.ez.point(self.arm.endeffector(), color='blue')



    def jacobian_loop(self):
        self.draw_arm(self.arm)
        self.draw_trajectory()

        self.current_point = self.trajectory.current()
        self.jacobian.control(self.current_point)

        self.errors.append(self.arm.error(self.current_point))
        self.draw_info()

        self.current_point = self.trajectory.next()

        self.ez.point(self.arm.endeffector(), color='blue')
```

```
def random_pose_loop(self):
    self.arm.set_random_pose()
    t1, t2 = self.arm.get_normalized_pose()
    self.arm2.set_normalized_pose(t1, t2)
    self.draw_arm(self.arm)
    self.draw_arm(self.arm2)

def generate_samples_loop(self):
    sample = self.generate_sample()
    vector = Point2(sample[2], sample[3])
    if vector.r > MIN_STEP and vector.r < MAX_STEP:
        norm_sample = self.normalize_sample(sample)
        self.dataset_writer.writerow(norm_sample)
        self.sample_counter += 1
    else:
        self.invalid_counter += 1

    if self.sample_counter % 10000 == 0:
        print "{} samples generated in this
            loop.".format(self.sample_counter)
        avg = self.sample_counter/float((self.sample_counter +
            self.invalid_counter))
        print "{}% valid".format(avg*100.)

    if self.sample_counter >= 2000000:
        self.exit()


def generate_sample(self):
    # generate random position
    self.arm.set_random_pose()

    #view
    # self.draw_arm(self.arm, color='blue')
    #endview

    t1, t2 = self.arm.t1, self.arm.t2
    ee1 = self.arm.endeffector()

    # generate random movement
    dt1 = normal_in_range(MIN_DT, MAX_DT)
    dt2 = normal_in_range(MIN_DT, MAX_DT)
```

```
        # dt1 = scale(0.0, 1.0, MIN_DT, MAX_DT, np.random.random())
        # dt2 = scale(0.0, 1.0, MIN_DT, MAX_DT, np.random.random())
        # move arm
        self.arm.move(dt1, dt2)
        ee2 = self.arm.endeffector()
        dee = ee2 - ee1

        #view
        # self.arm2.t1, self.arm2.t2 = self.arm.t1, self.arm.t2
        # self.draw_arm(self.arm2, color='red')
        # print [t1,t2, dee.x, dee.y, dt1, dt2], ":", ee1, " ->", ee2
        # raw_input()
        #endview

        return [t1,t2, dee.x, dee.y, dt1, dt2]

    def normalize_sample(self, sample):
        t1  = scale(MIN_THETA, MAX_THETA, 0.0, 1.0, sample[0])
        t2  = scale(MIN_THETA, MAX_THETA, 0.0, 1.0, sample[1])
        dx  = scale(-MAX_STEP, MAX_STEP, 0.0, 1.0, sample[2])
        dy  = scale(-MAX_STEP, MAX_STEP, 0.0, 1.0, sample[3])
        dt1 = scale(MIN_DT, MAX_DT, 0.0, 1.0, sample[4])
        dt2 = scale(MIN_DT, MAX_DT, 0.0, 1.0, sample[5])

        norm_sample = [t1, t2, dx, dy, dt1, dt2]
        for value in norm_sample:
            assert value >= 0.0 and value <= 1.0
        return norm_sample


    def draw_arm(self, arm, color='black'):
        p1 = Point2().polar(arm.l1, arm.t1)
        p2 = Point2().polar(arm.l2, arm.t2+arm.t1) + p1
        self.ez.line(Point2(), p1, color=color)
        self.ez.point(p1)
        self.ez.line(p1, p2, color=color)
        self.ez.point(p2)

    def draw_trajectory(self):
        self.ez.lines(self.trajectory.points)

    def draw_info(self):
```

```
        # tamanho do passo atual
        val = self.traj_step
        val_text = "Step size.....{0}".format(val)
        self.ez.text(val_text, Point2(10,10), onScreen=True)
        # passo atual
        val = self.trajectory.curr_idx()
        val_text = "Current step..{0}".format(val)
        self.ez.text(val_text, Point2(10,20), onScreen=True)
        # erro medio
        val = self.mse()
        val_text = "Avg. error....{0}".format(val)
        self.ez.text(val_text, Point2(10,30), onScreen=True)

        if self.loop == JACOBIAN:
            # temporary
            if len(self.plot_errors) > 2:
                self.ez.lines(self.plot_errors,color='blue')

    def mse(self):
        squared = [e*e for e in self.errors]
        # squared = [e for e in self.errors]
        return sum(squared)/float(len(squared))

    def run(self):
        self.ez.run()

    def reset(self):
        mse = self.mse()
        print self.traj_step, mse

        if self.loop == TEST_NETWORK:
            self.net_logger.add_data(self.traj_step, mse)
        elif self.loop == JACOBIAN:
            self.jacobian_logger.add_data(self.traj_step, mse)


        self.traj_step += TRAJ_STEP_INC
        if self.traj_step > TRAJ_STEP_MAX:
            self.exit()

        self.trajectory = Trajectory(
            step_length=self.traj_step,
```

```
            restart=self.reset)
        self.errors = []
        self.arm = Arm()

        if self.loop == JACOBIAN:
            self.jacobian.arm = self.arm
            # temporary
            self.plot_errors.append(
                Point2(self.traj_step * 10, mse*1000)
            )
        elif self.loop == TEST_NETWORK:
            self.net.arm = self.arm

    def exit(self):
        if self.loop == TEST_NETWORK:
            self.net_logger.plot("Step Size", "MSE")
        if self.loop == JACOBIAN:
            self.jacobian_logger.plot("Step Size", "MSE")
        raw_input()
        exit()


if __name__ == '__main__':
    app = Application()
    app.run()
```

A.2 – *ARM.PY*

```
from point2 import Point2
import numpy as np
from helper import scale

MAX_THETA = np.pi
MIN_THETA = -np.pi

class Arm:
    def __init__(self, l1=300, l2=300, t1=0.0, t2=np.pi/2.):
        self.t1 = t1
        self.t2 = t2
```

```
        self.l1 = l1
        self.l2 = l2

    def move(self, dt1, dt2):
        self.t1 += dt1
        self.t2 += dt2

        if self.t1 > MAX_THETA:
            self.t1 -= 2*np.pi
        if self.t2 > MAX_THETA:
            self.t2 -= 2*np.pi

        if self.t1 < MIN_THETA:
            self.t1 += 2*np.pi
        if self.t2 < MIN_THETA:
            self.t2 += 2*np.pi

    def endeffector(self):
        p1 = Point2().polar(self.l1, self.t1)
        return Point2().polar(self.l2, self.t2+self.t1) + p1

    def set_random_pose(self):
        t1 = scale(0.0, 1.0, MIN_THETA, MAX_THETA, np.random.random())
        t2 = scale(0.0, 1.0, MIN_THETA, MAX_THETA, np.random.random())
        # check if the values are between MIN_THETA and MAX_THETA
        assert t1 > MIN_THETA and t1 < MAX_THETA
        assert t2 > MIN_THETA and t2 < MAX_THETA
        # if they are, set joints to the values
        self.t1 = t1
        self.t2 = t2

    def get_normalized_pose(self):
        t1 = scale(MIN_THETA, MAX_THETA, 0.0, 1.0, self.t1)
        t2 = scale(MIN_THETA, MAX_THETA, 0.0, 1.0, self.t2)
        # check if the values are between 0.0 and 1.0
        assert t1 > 0.0 and t1 < 1.0
        assert t2 > 0.0 and t2 < 1.0
        return t1, t2

    def set_normalized_pose(self, t1, t2):
        t1 = scale(0.0, 1.0, MIN_THETA, MAX_THETA, t1)
        t2 = scale(0.0, 1.0, MIN_THETA, MAX_THETA, t2)
```

```
        # check if the values are between MIN_THETA and MAX_THETA
        assert t1 > MIN_THETA and t1 < MAX_THETA
        assert t2 > MIN_THETA and t2 < MAX_THETA
        # if they are, set joints to the values
        self.t1 = t1
        self.t2 = t2


    def error(self, target):
        # returns the euclidian distance between the endeffector and the target
        return (self.endeffector() - target).r
```

A.3 – *HELPER.PY*

```
import numpy as np

# units are in milimeters
# TRAJ_STEP_MIN = 0.1
# TRAJ_STEP_MAX = 20.
# TRAJ_STEP_INC = 0.1
TRAJ_STEP_MIN = 0.1
TRAJ_STEP_MAX = 100.
TRAJ_STEP_INC = 0.1

MIN_STEP = 0.1
MAX_STEP = 100.

DT_VAL = 20
MIN_DT = np.radians(-DT_VAL)
MAX_DT = np.radians(DT_VAL)



def scale(min_ini, max_ini, min_fin, max_fin, x):
    # rescale x from a initial range to a final range
    p = float(max_fin - min_fin)/float(max_ini - min_ini)
    d = min_fin - (min_ini * p)
    return np.float64((x*p) + d)
```

```
def normal_in_range(ini, fin):
    r = -10
    while r < -1 or r > 1: # or r == 0.0:
        r = np.random.normal(0, 0.33, 1)[0]

    # if r > 0.:
    #     r = 1 - r
    # else:
    #     r = -1 - r
    return scale(-1.0, 1.0, ini, fin, r)
```

A.4 – *JACOBIAN.PY*

```
from point2 import Point2
import numpy as np
from arm import Arm

class JacobianControler:
    def __init__(self, arm=Arm()):
        self.arm = arm

    def control(self, target):
        curr_end = self.arm.endeffector()
        delta_pos = target - curr_end

        iJ = 1/(self.arm.l1*self.arm.l2*np.sin(self.arm.t2)) * np.array([
        [self.arm.l2*np.cos(self.arm.t1 + self.arm.t2),
         self.arm.l2*np.sin(self.arm.t1 + self.arm.t2)],
        [-self.arm.l1*np.cos(self.arm.t1)-self.arm.l2*np.cos(self.arm.t1+self.arm.t2),
         -self.arm.l1*np.sin(self.arm.t1)-self.arm.l2*np.sin(self.arm.t1+self.arm.t2)]
        ])

        delta_joints = iJ.dot(np.array([delta_pos.x, delta_pos.y]))

        # self.arm.t1 += delta_joints[0]
        # self.arm.t2 += delta_joints[1]
        self.arm.move(delta_joints[0], delta_joints[1])
```

A.5 – *LOGGER.PY*

```
import numpy as np
import matplotlib.pyplot as plt
import csv


class Logger:
    def __init__(self, name):
        self.x = []
        self.y = []
        self.name = name
        csvpath = 'reports/{}.csv'.format(name)
        csvfile = open(csvpath, 'wb')
        self.csvwriter = csv.writer(csvfile)

    def add_data(self, x, y):
        self.x.append(x)
        self.y.append(y)
        self.csvwriter.writerow([x, y])

    def plot(self, xlabel, ylabel, name=-1):
        if name == -1:
            name = self.name
        plt.plot(self.x, self.y)
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        image_path = 'reports/{}.png'.format(name)
        plt.savefig(image_path)
```

A.6 – *NET.PY*

```
from point2 import Point2
import numpy as np
from arm import *
from keras.models import load_model
from helper import *
```

```
class Net:
    def __init__(self, modelid, arm=Arm()):
        self.arm = arm
        model_file = "nets/model{}.h5".format(modelid)
        self.model = load_model(model_file)

    def control(self, target):
        nt1, nt2 = self.arm.get_normalized_pose()
        curr_end = self.arm.endeffector()
        delta_pos = target - curr_end

        dx = scale(-MAX_STEP, MAX_STEP, 0.0, 1.0, delta_pos.x)
        dy = scale(-MAX_STEP, MAX_STEP, 0.0, 1.0, delta_pos.y)

        # feed the net
        net_in = np.array([nt1, nt2, dx, dy])
        net_out = self.model.predict(net_in.reshape(1,4))[0]
        dt1 = scale(0.0, 1.0, MIN_DT, MAX_DT, net_out[0])
        dt2 = scale(0.0, 1.0, MIN_DT, MAX_DT, net_out[1])
        assert dt1 >= MIN_DT and dt1 <= MAX_DT
        assert dt2 >= MIN_DT and dt2 <= MAX_DT

        self.arm.move(dt1, dt2)
```

A.7 – *TRAIN_NET.PY*

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import csv

import time
import datetime

EPOCHS = 500
SAVING_STEP = 10
```

```
# load dataset
dataset = np.loadtxt("dataset_15_20-50.csv", delimiter=',')
# split into X and Y
n_inputs = 4
n_outputs = 2
X = dataset[:,0:n_inputs]
Y = dataset[:,n_inputs:]
# split into train and test
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.1)



model = Sequential()

layer_sizes = [128, 64, 36, 16 ,8]
layer_activations = ['relu', 'relu','relu', 'relu', 'relu']

for i, l in enumerate(layer_sizes):
    if i == 0:
        model.add(Dense(l, activation=layer_activations[i],
            input_shape=(n_inputs,)))
    else:
        model.add(Dense(l, activation=layer_activations[i]))

model.add(Dense(n_outputs, activation='sigmoid'))

model.summary()
model.compile(loss='mean_squared_error', optimizer='adam') #, metrics=['mae',
    euclid_dist])


loss = []
val_loss = []


ts = time.time()
st = datetime.datetime.fromtimestamp(ts).strftime('%Y%m%d_%H%M%S')
csvreport = open('nets/{}.csv'.format(st), 'wb')
csvwriter = csv.writer(csvreport)
for epoch in range(EPOCHS):
    history = model.fit(x_train, y_train,
```

```
        batch_size=64,
        epochs=1,
        # validation_split=0.2,
        verbose=1,
        validation_data=(x_test,y_test)
    )

    loss += history.history['loss']
    val_loss += history.history['val_loss']

    print "Epoch: {} - loss: {} - val_loss: {}".format(epoch, loss[-1],
        val_loss[-1])

    if epoch % SAVING_STEP == 0:
        print "saving model with", epoch, "epochs"
        model.save('nets/model{}.h5'.format(st))
        csvwriter.writerow([loss, val_loss])
        plt.figure(figsize=(5,5))
        plt.plot(np.arange(1, len(loss)+1), loss)
        plt.plot(np.arange(1, len(val_loss)+1), val_loss)
        # plt.show()
        image_path = "nets/trained{}.png".format(st)
        plt.savefig(image_path)

print "saving final model with", epoch, "epochs"
model.save('nets/model{}.h5'.format(st))

plt.figure(figsize=(8,8))
plt.plot(np.arange(1, len(loss)+1), loss)
plt.plot(np.arange(1, len(val_loss)+1), val_loss)
# plt.show()
image_path = "nets/trained{}.png".format(st)
plt.savefig(image_path)
```

A.8 – *TRAJECTORY.PY*

```
from point2 import Point2
import numpy as np
```

```
TRAJ_CENTER = Point2(200, 300)

class Trajectory:
    def __init__(self, step_length=0.1, radius=100, points_count=1000,
        center=TRAJ_CENTER, restart=None):
        self.points = []
        self.counter = 0
        self.center = center
        self.radius = radius
        self.step_length = step_length
        self.points_count = points_count
        self.restart_function = restart

        self.generate()

    def generate(self):
        p = Point2(self.radius, 0)
        angle_step = 2*np.arcsin(self.step_length/(2.*self.radius))
        while len(self.points) < self.points_count:
            self.points.append(Point2(self.center+p))
            p.a += angle_step

    def current(self):
        return self.points[self.counter]

    def curr_idx(self):
        return self.counter

    def next(self):
        point = self.points[self.counter]
        new_counter = self.counter+1
        if new_counter > len(self.points)-1:
            new_counter = 0
            if self.restart_function:
                self.restart_function()

        self.counter = new_counter
        return point
```