Universidade de São Paulo - USP
Instituto de Ciências Matemáticas e de Computação - ICMC
Departamento de Sistemas de Computação - SSC

# Projeto Final

## Controle Preditivo Explícito a partir de Perceptron Multicamadas (MLP)

**Inteligência Artificial Embarcada — SSC0967**

**Artur Brenner Weber** — Número USP: 12675451
**Carlos Henrique Craveiro Aquino Veras** — Número USP: 12547187
**Vicenzo D'Arezzo Zilio** — Número USP: 13671790

Prof. Dr. Vanderlei Bonato
Engenharia/Ciência de Computação

10 de dezembro de 2025

# Sumário

# 1. Introdução

A automação de sistemas robóticos complexos, particularmente Veículos Aéreos Não Tripulados (VANTs), exige estratégias de controle capazes de lidar com dinâmicas rápidas e restrições físicas severas. O Controle Preditivo Baseado em Modelo (MPC - *Model Predictive Control*) estabeleceu-se como uma técnica de referência para tais aplicações, oferecendo robustez e otimalidade ao resolver problemas de otimização restritos em tempo real[1]. No entanto, a carga computacional associada à resolução iterativa de Programação Quadrática (QP) impõe barreiras significativas para a implementação em hardware embarcado de baixo custo e consumo energético[1].

Neste contexto, a interseção entre a Teoria de Controle e a Inteligência Artificial (IA) surge como uma fronteira promissora. A capacidade de Redes Neurais Profundas (*Deep Neural Networks*) de aproximar funções complexas com tempos de inferência determinísticos sugere um caminho para acelerar os controladores clássicos. Este paradigma híbrido, onde a IA auxilia, mas não substitui totalmente o garantidor de estabilidade clássico, é o foco deste trabalho, buscando entender e integrar subsistemas baseados em aprendizado ao pipeline tradicional de controle preditivo.

## 1.1 Objetivos

O objetivo principal deste projeto, desenvolvido no âmbito da disciplina de Inteligência Artificial Embarcada, é implementar e validar uma estratégia de *Neural Warm Start* para um controlador MPC aplicado ao pouso de um quadrotor.

Os objetivos específicos incluem:

1. Replicar e adaptar a metodologia proposta por Chen et al. (2022) [1] para o treinamento de uma rede neural capaz de predizer as variáveis primais ótimas de um problema de controle;

2. Desenvolver um *pipeline* completo de *Machine Learning*, desde a geração de dados através da simulação da dinâmica até o treinamento supervisionado com funções de perda especializadas.

3. Portar o modelo treinado para um sistema embarcado representativo (Raspberry Pi 4B), utilizando ferramentas de otimização como ONNX Runtime e quantização.

4. Avaliar o desempenho em malha fechada, comparando métricas de latência, número de iterações do solucionador e precisão de trajetória entre a abordagem híbrida e o método clássico.

## 1.2 Organização do Trabalho

Este relatório está estruturado da seguinte forma: O Capítulo 2 apresenta o Contexto e a Motivação, detalhando o problema do pouso de drones e a base teórica do artigo de referência. O Capítulo 3 descreve a Aplicação do Artigo, focando no pré-processamento, na evolução das funções de perda (*Loss Functions*) e na arquitetura da rede neural. O Capítulo 4 aborda a Otimização da Rede, discutindo a escolha de hiperparâmetros e técnicas de quantização. O Capítulo 5 detalha o *Deploy* da Rede, descrevendo a integração com o sistema em C++ e a análise dos resultados experimentais. Por fim, o Capítulo 6 apresenta as conclusões e perspectivas futuras.

# 2. Contexto e Motivação

## 2.1 Contexto do Projeto

O presente trabalho foi desenvolvido no âmbito da disciplina de Inteligência Artificial Embarcada, visando a aplicação prática de técnicas de aprendizado profundo (*Deep Learning*) em sistemas de controle. O cenário de aplicação escolhido baseia-se em um problema real de robótica aérea: o pouso autônomo de um veículo aéreo não tripulado (VANT), especificamente um quadrotor, em condições adversas, como o pouso em plataformas móveis ou em ambientes marítimos (*automar*).

Este cenário específico deriva de uma pesquisa de iniciação científica previamente conduzida pelo integrante do grupo, Carlos Henrique Craveiro, onde a estratégia de controle adotada foi o Controle Preditivo Baseado em Modelo (MPC - *Model Predictive Control*). No contexto deste projeto, o MPC atua como o "cérebro" da aeronave. O controlador deve, a todo instante, dado o estado atual do sistema (posição, velocidade, orientação), calcular a sequência de ações ótimas para os atuadores (motores) de forma a conduzir o drone até o alvo final (o solo ou convés de um navio), respeitando restrições físicas e dinâmicas.

A técnica de MPC formula o problema de controle como um problema de otimização matemática (Programação Quadrática - QP) que deve ser resolvido a cada passo de tempo discreto. Diferente de controladores clássicos como o PID, que reagem ao erro passado, o MPC "olha para frente" (horizonte de predição), antecipando o comportamento futuro do sistema. Naturalmente, a metodologia preditiva é consideravelmente mais complexa, tanto por escolher uma abordagem deliberativa quanto por estender a análise a momentos futuros. Logo, em sistemas mais complexos, como drones multiarticulados ou manipuladores aéreos, a dimensão do problema pode facilmente escalar para a ordem de milhares de variáveis, tornando a resolução computacional extremamente onerosa.

A implementação proposta visa portar essa inteligência para a plataforma de computação embarcada Raspberry Pi 4B, equipada com 8 GB de memória RAM. O objetivo é estabelecer uma comunicação em malha fechada entre a simulação física do drone (executada externamente) e o controlador neural embarcado no hardware, validando a capacidade de processamento dos quatro núcleos ARM Cortex-A72 presentes no SoC Broadcom BCM2711.

## 2.2 Motivação e Desafios de Tempo Real

A principal motivação para a integração de Redes Neurais neste fluxo de controle reside no custo computacional proibitivo do MPC para aplicações embarcadas rápidas. Embora o MPC ofereça

robustez e tratamento de restrições, a necessidade de resolver um problema de otimização complexo a cada milissegundo impõe uma barreira técnica significativa.

Em aplicações de robótica aérea, o requisito de tempo real é necessário (*soft real-time*). O algoritmo de controle é responsável pela estabilização ativa da aeronave. Se o solucionador (*solver*) de otimização não entregar uma resposta em torno de um período de amostragem estipulado, o sistema torna-se instável, podendo resultar na queda catastrófica do drone. Diferentemente de resoluções offline, onde o tempo de processamento é tolerável, em sistemas embarcados, a latência é um fator crítico de segurança.

A abordagem motivadora deste trabalho é a substituição ou auxílio do processo de otimização tradicional por uma Rede Neural Profunda (Perceptron Multicamadas). A hipótese central é que uma rede neural bem treinada pode aproximar a complexa lei de controle do MPC com um tempo de inferência muito menor e, crucialmente, determinístico. Enquanto um solver iterativo pode variar seu tempo de execução dependendo da complexidade do estado atual, uma rede neural (como uma MLP) possui um custo computacional fixo e previsível, dependendo apenas do número de operações da arquitetura.

No entanto, redes neurais puras não oferecem garantias de otimalidade ou factibilidade - essenciais na solução de problemas de otimização. Portanto, a motivação evolui para o uso de uma arquitetura híbrida: utilizar a Rede Neural para fornecer um Warm Start ao solucionador (*solver*). A seleção do algoritmo de otimização é crítica neste contexto, devendo-se priorizar métodos capazes de acelerar a convergência a partir de soluções subótimas, como os solvers de conjuntos ativos (*active set solvers*). Se a rede tiver um bom desempenho e o solver for adequado para warm-starting, reduz-se drasticamente o número de iterações necessárias, economizando recursos computacionais e viabilizando o MPC em hardware embarcado limitado. Como mencionado na definição do escopo, é aceitável que a rede cometa pequenos erros, desde que a estimativa permaneça na vizinhança da solução ótima, permitindo o refinamento final pelo solver.

# 3. O Artigo Base: Metodologia e Adaptações

## 3.1 Visão Geral da Abordagem Híbrida

A fundamentação teórica e metodológica deste projeto baseia-se no artigo *"Large Scale Model Predictive Control with Neural Networks and Primal Active Sets"* [1]. Este trabalho propõe um procedimento "Explícito-Implícito" para computar a lei de controle do MPC com garantias de viabilidade recursiva e estabilidade assintótica.

A arquitetura central consiste em combinar uma rede neural totalmente conectada (*Fully Connected Neural Network*), treinada offline, com um solucionador de *Primal Active Set* executado online. O papel da rede neural é fornecer uma inicialização da variável de decisão primal ($z$) para o solver.

No MPC convencional (implícito), o solver inicia a busca pela solução ótima a partir de um "Cold Start" (geralmente a solução do passo anterior deslocada ou um vetor nulo). O artigo demonstra que, para sistemas de larga escala, essa abordagem é ineficiente. A proposta do artigo utiliza a rede neural para mapear o estado atual do sistema ($x$) diretamente para uma aproximação das variáveis de otimização ($z$). Como a rede neural sozinha não pode garantir a satisfação estrita das restrições de segurança (barreiras estaduais e de entrada), sua saída é utilizada apenas como um ponto de partida privilegiado.

## 3.2 Arquitetura da Rede e Função de Perda Primal-Dual

Para a disciplina de IA Embarcada, o foco principal recai sobre a estrutura e o treinamento da rede neural. O artigo utiliza uma Rede Neural Profunda (DNN) com ativação ReLU. A escolha da ReLU é estratégica: como a solução explícita de um problema de MPC linear é uma função linear por partes (*piecewise affine*) sobre politopos, uma rede ReLU possui a capacidade teórica de representar exatamente essa função, dada profundidade e largura suficientes.

Um dos pontos mais inovadores do artigo [1], que replicamos neste projeto, é a formulação da função de perda (*loss function*). Com a necessidade de atender a um problema de otimização, é necessário que a rede aprenda não somente a encontrar um valor ótimo mas também a atender às restrições. Portanto, é introduzida uma nova função de perda baseada em informações primais (que representam a otimalidade do valor) e também duais (que representam a factibilidade do valor).

A função proposta consiste no cálculo do Erro Quadrático Médio (MSE) do Lagrangiano do problema para o primal predito e o primal correto, fornecido pelo conjunto de dados. O

Lagrangiano de um problema de otimização restrito é construído pela incorporação das restrições ao problema original por meio dos multiplicadores de Lagrange. Essa construção permite definir o problema dual, no qual a função objetivo já é suficiente para garantir otimalidade (menor valor da função) e factibilidade (atendimento de todas as restrições).

$$l(\theta) := \sum_{i=1}^{|\mathcal{D}|} (\mathcal{L}(\tilde{\pi}(x_i|\theta), \nu_i^*, \lambda_i^*|x_i) - \mathcal{L}(z_i^*, \nu_i^*, \lambda_i^*|x_i))^2 \tag{3.1}$$

Na definição acima, $\mathcal{L}$ é o Lagrangiano associado ao problema de Programação Quadrática (QP), $\tilde{\pi}(x|\theta)$ é a predição da rede, e $(z^*, \nu^*, \lambda^*)$ são as variáveis primais e duais ótimas. Essa abordagem permite que o treinamento seja supervisionado não apenas pela solução de controle, mas também pelas restrições ativas e pela geometria do problema de otimização, resultando em uma aproximação que tende a ser primal-viável com mais frequência.

## 3.3 Garantias de Estabilidade e Otimização Online

O framework proposto por [1] não confia cegamente na IA. A rede neural fornece o *Warm Start*, mas o garantidor de estabilidade é o solver de *Active Set*. O processo ocorre em duas fases:

- **Fase I (Viabilidade):** O solver recebe a estimativa da rede neural e, caso ela viole alguma restrição (ex: velocidade máxima do drone excedida), projeta a solução de volta para a região viável.
- **Fase II (Otimicidade):** A partir de uma solução viável, o solver itera para reduzir o custo até que um critério de sub-otimalidade seja atingido.

Uma contribuição chave replicada no conceito do projeto é o "Early Termination" (Término Antecipado). Como a rede neural já fornece uma solução muito próxima da ótima, o solver não precisa iterar até a convergência absoluta. O artigo define um certificado de sub-otimalidade baseado na lacuna de dualidade (*duality gap*). Assim que a solução atinge um nível aceitável de qualidade (garantido pela teoria de Lyapunov para estabilidade), o solver é interrompido. Isso economiza ciclos preciosos da CPU do sistema embarcado Aquila AM69.

## 3.4 Adaptações e Realidade do Projeto

Conforme relatado na apresentação do projeto, a replicação integral da metodologia de [1] apresentou desafios práticos significativos. Os autores originais customizaram profundamente o solver QP, desmontando-o para acessar iterações internas de Fase I e Fase II.

Para a nossa prova de conceito na disciplina, realizamos simplificações estratégicas. Em vez de reescrever um solver de otimização do zero, optamos por acoplar a rede neural treinada (seguindo a metodologia de perda Lagrangiana e geração de dados do artigo) a um solver

comercial ou open-source padrão, passando a saída da rede como ponto de partida. Embora perca-se a granularidade do controle sobre as fases internas, essa abordagem mantém o benefício macro do *Warm Start*.

O foco do nosso trabalho desloca-se, portanto, para a eficiência da inferência da rede neural no hardware alvo. Buscamos otimizar a execução da MLP na CPU da Rasberry Pi 4B, avaliando o trade-off entre o tamanho da rede (número de neurônios e camadas), a precisão da predição do controle ótimo e o tempo total de resposta do sistema em malha fechada. Esta validação prática visa demonstrar que a IA Embarcada pode atuar como um catalisador para tecnologias de controle avançado que, de outra forma, permaneceriam restritas a simulações em desktops potentes.

# 4. Método

Durante o desenvolvimento, a versão inicial do método acabou por não funcionar. Nesse sentido, serão apresentadas duas versões diferentes de pré-processamento e função de perda: **inicial** - mais complexa, que foi substituída pela segunda; **final** - mais simples, versão que convergiu e foi utilizada no trabalho.

**Todo o código fonte mencionado por ser encontrado no repositório git:** *Referência Código Fonte.*

## 4.1 Geração do Conjunto de dados

Em primeira instância, apresenta-se a formulação do problema de controle como *Box-constrained Quadratic Programming*, onde a variável $z$ representa, respectivamente, o estado estimado e a atuação calculada para cada momento no horizonte. $P$ e $q$ são a matriz e o vetor que definem a função objetivo, com base no modelo do sistema, enquanto $A$ representa as restrições dinâmicas e operacionais do sistema. Por fim, $l$ e $u$ são os limites inferiores e superiores calculados para $Az$.

$$\min_z \frac{1}{2} z^T P z + q^T z \quad \text{s.t.} \quad l \leq Az \leq u \tag{4.1}$$

Com essa definição e um solver, é possível resolver o problema de otimização e criar a variável de atuação ótima que o sistema dinâmico recebe. Portanto, para simular o cenário de controle, foram integrados em uma arquitetura SIL:

- **Estimação e Objetivo:** processamento da informação da aeronave em relação a um contexto, produzindo o estado inicial do sistema e seu objetivo local a ser atingido;
- **Controle por Solver:** com o estado e o objetivo bem estipulados, definição e solução do problema de controle para gerar o sinal de atuação ótimo;
- **Modelagem Dinâmica:** simulação do comportamento do sistema ao receber os sinais de atuação em dado contexto de mundo, possibilitando a continuidade do loop de simulação.

A partir da ferramenta desenvolvida, em diferentes trajetórias, para cada ciclo de controle, coletou-se informações de: estado inicial ($x$) ; primal calculado pelo solver ($z^*$) ; duais calculados pelo solver ($\nu^*$ e $\lambda^*$); limites inferiores e superiores das restrições ($l$ e $u$). Enfim, utilizou-se esses dados para construir uma base para o aprendizado da rede, com o objetivo de aproximar o valor ótimo obtido pelo solver.

## 4.2 Pré-processamento

Para a solução de um problema de otimização através de redes neurais, há duas grandes preocupações em relação aos dados. Para cada nuance abaixo, são aplicadas transformações com o objetivo de equilibrar a representatividade dos dados e o desempenho do treinamento do modelo.

1. **Distribuição de Probabilidade dos Dados:** muitos algoritmos de aprendizado assumem que os dados seguem distribuições aproximadamente normais, preferencialmente centradas em zero, com desvio padrão igual a 1 e livres de outliers. Para atender a essas condições, aplicam-se transformações específicas que tornam a distribuição original — muitas vezes heterogênea — mais próxima desse formato;

2. **Condicionalidade das matrizes:** em muitos problemas numéricos, especialmente os que envolvem inversão de matrizes, sistemas lineares e otimização, o desempenho e a precisão dos algoritmos dependem fortemente do condicionamento da matriz envolvida. Matrizes mal condicionadas possuem número de condição elevado, o que amplifica erros de arredondamento e torna as soluções numericamente instáveis. Para mitigar esses efeitos, empregam-se técnicas como regularização, reescalonamento das variáveis e decomposições numéricas adequadas, visando melhorar a estabilidade computacional do problema.

### 4.2.1 Versão Inicial

Inicialmente, com a abordagem mais complexa, a função de perda incluia o cálculo do lagrangiano completo. Logo, neste momento, preocupou-se principalmente com a redução da alta dimensionalidade da matriz $P$. Paralelamente, também foi feita a normalização dos dados de entrada da função de perda e da rede.

#### 4.2.1.1 Redução da Condicionalidade da Matriz Quadrática

Trasnformações aplicadas:

1. **Regularização de Tikhonov:**consiste em adicionar um termo à diagonal da matriz P, o que desloca seus autovalores para longe de zero. Essa operação melhora o número de condição e evita instabilidades numéricas durante inversões ou operações matriciais sensíveis.

2. **Normalização Espectral das Matrizes:** ajusta a matriz $P$ de modo que seu maior autovalor (ou norma espectral) seja igual a um valor pré-definido, tipicamente 1. Ao controlar a amplitude do espectro, reduz-se a disparidade entre autovalores, melhorando o condicionamento.

3. **Escalonamento das Matrizes:** aplica fatores de escala às linhas e/ou colunas de $P$, normalmente para que cada variável tenha magnitude comparável. Esse procedimento diminui diferenças de ordem de grandeza que poderiam causar mau condicionamento e instabilidade em algoritmos numéricos. A escala foi definida como o inverso da raiz quadrada de $P$.

#### 4.2.1.2 Regularização da Distribuição dos dados

Com o objetivo de ajustar a distribuição de probabilidade e o formato de apresentação dos dados à função de perda e ao modelo, foram aplicadas **Normalizações por z-scalling**:

$$x' = \frac{x - \mu_x}{\sigma_x} \tag{4.2}$$

Com essa operação, garantimos que os dois primeiros momentos estatísticos — média e variância — coincidam com os da distribuição gaussiana padrão. Foram normalizados: estado inicial ($x$); primal ($z$); duais ($\nu$ e $\lambda$).

Abaixo, segue a distribuição inicial dos dados - mais visualizações dos dados, antes e depois do pré-processamento, podem ser encontradas no código fonte.

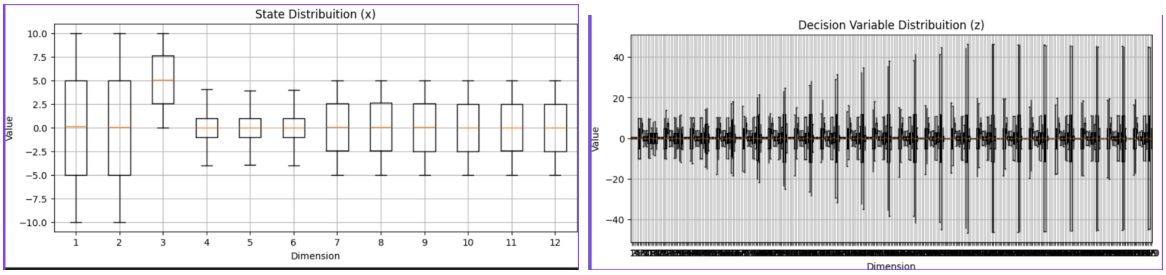


Figura 4.1: Distribuição dos estados iniciais e trajetórias ótimas geradas para treinamento.

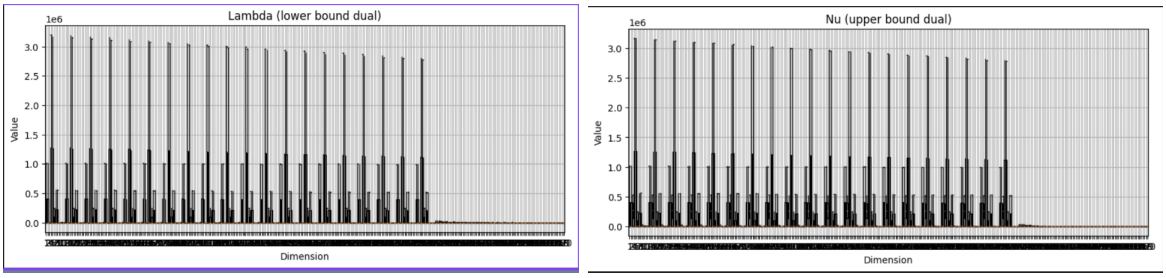


Figura 4.2: Visualização das superfícies de custo e restrições associadas ao problema QP original.

### 4.2.2 Versão Final

Para o caso da versão final, não utilizou-se mais a matriz ($P$), o vetor ($q$) e os multiplicadores duais ($\nu$ e $\lambda$). Portanto, as transformações de condicionalidade não se fizeram mais necessárias,

assim como a normalização dos multiplicadores duais.

## 4.3 Função de Perda

A função de perda, também referida como *loss*, é uma função escalar que deve ser responsável por verificar a qualidade da aproximação feita pela rede neural. Esse cálculo é essencial para a convergência de um modelo, dado que o gradiente da função de perda em relação aos parâmetros da rede é responsável por atuaizar iterativamente os pesos e viéses do modelo, aproximando o a predição da expectativa. Portanto, uma função de perda adequada deve ser escalar, contínua por partes e numericamente estável, de modo a permitir avaliação consistente e otimização eficiente.

### 4.3.1 Função Inicial

O objetivo inicial era replicar exatamente a função proposta pelo artigo 3.2. Um único detalhe foi adicionado: ao invés de tentar usar o lagrangiano clássico, foi utilizado o lagrangiano aumentado. Essa decisão é recomendada para aumentar a relação entre o valor de loss e o atendimento das condições KKT. Segue a definição utilizada para o lagrangiano aumentado de um problema de otimização, conforme a formatação utilizada 4.1.

$$\begin{aligned}\mathcal{L}_\rho(z,\nu,\lambda) \;=& \frac{1}{2}z^\top Hz + f^\top z \;+\; \nu^\top(Az-u) \;+\; \lambda^\top(\ell - Az) \\ &+\; \frac{\rho}{2}\left\|\max(0,\; Az-u)\right\|_2^2 \;+\; \frac{\rho}{2}\left\|\max(0,\; \ell - Az)\right\|_2^2\end{aligned} \tag{4.3}$$

#### 4.3.1.1 Problemas Encontrados

Diversos problemas foram observados com a função de perda proposta. Para investigar o comportamento do treinamento, foram realizados os seguintes testes:

1. **Correlação entre valor da loss e norma do gradiente para entradas aleatórias:** Foi observada correlação consistente entre o valor da função de perda e a magnitude do gradiente, indicando coerência matemática e implementação correta do cálculo dos termos da loss. (*Resultado: OK*)

2. **Teste de convergência em cenário de overfitting (capacidade de aprendizado):** O modelo demonstrou capacidade de convergir e ajustar-se a um conjunto reduzido de dados, confirmando que a arquitetura e o otimizador são capazes de minimizar a função de perda em condições ideais. (*Resultado: OK*)

3. **Análise da distribuição de probabilidade dos valores da loss para entradas aleatórias:** Este teste revelou comportamento instável da função de perda. Observou-se um valor esperado elevado, porém relativamente estável, acompanhado de explosões frequentes no valor total da loss, principalmente associadas aos termos quadráticos, duais e do Lagrangiano aumentado.

- Valor esperado da loss elevado, porém com baixa variância local;
- Explosões esporádicas de grande magnitude, dominadas pelos termos quadráticos e duais;
- Comportamento não robusto mesmo após aplicação de pré-processamentos e balanceamento entre os diferentes termos da função de perda.

(*Resultado: Não OK*)

Por fim, mesmo após a aplicação de diferentes estratégias de normalização, pré-processamento e ponderação dos termos constituintes da função de perda, não foi possível eliminar o comportamento instável observado. Segue alguns gráficos retirados desse experimento, onde observa-se a explosão do valor da função diversas vezes ao longo de longos testes.

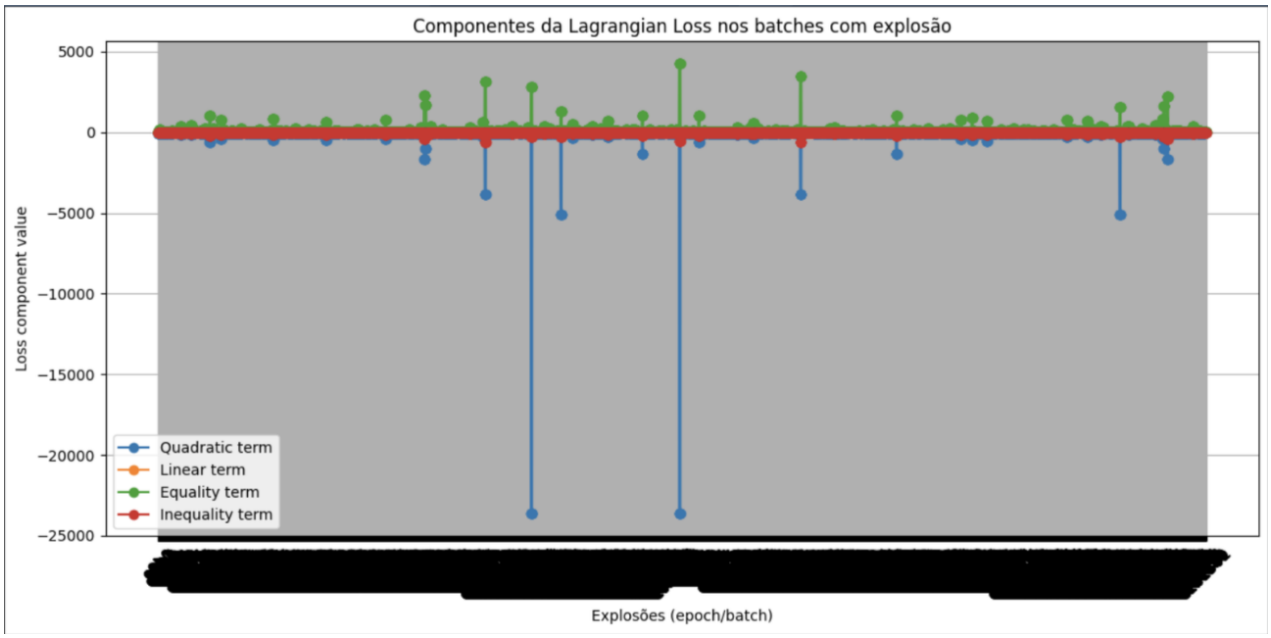


Figura 4.3: Visualização da maginitude ao longo das amostras dos diferentes termos envolvidos no cálculo do lagrangiano.

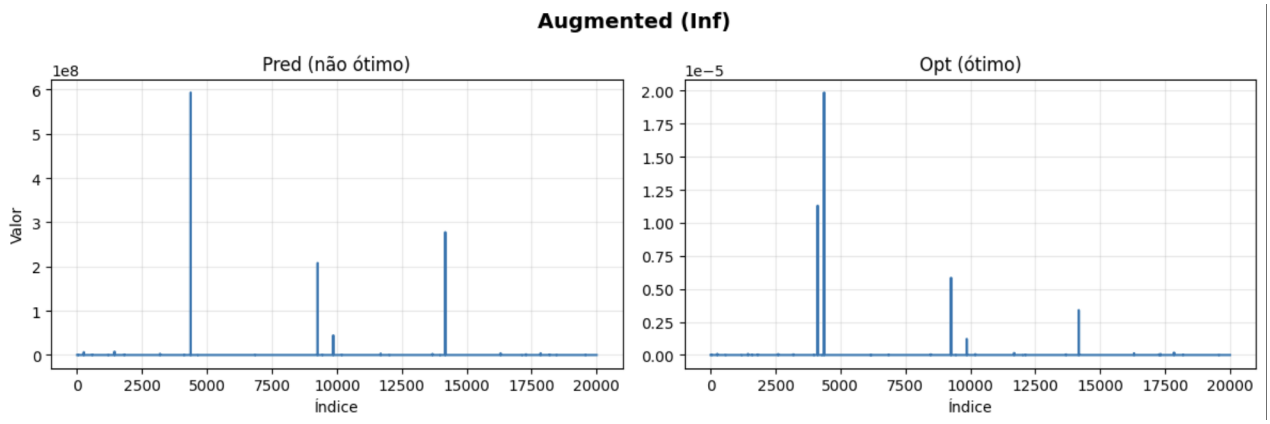


Figura 4.4: Visualização comparativa da magitude do termo do lagrangiano - referente às restrições inferiores - para os valores ótimos e os valores não ótimos coletados.

### 4.3.2 Função Final

Com o objetivo de garantir a convergência da rede, foi criada uma segunda função de perda capaz de emular o comportamento da anterior de maneira mais simples e estável. Para isso, o cálculo complexo do lagrangiano foi substituído por uma simples subtração dos primais e a adição dos termos do lagrangiano aumentado referentes ao atendimento das restrições. Com esses dois componentes, é possível garantir que a loss utilizada inclui tanto a significância do princípio de otimalidade - minimiza a diferença primal - quanto do princípio de factibilidade - minimiza o erro associado ao não atendimento das restrições.

$$\ell(\theta) := \sum_{i=1} \Big( (\tilde{\pi}(x_i \mid \theta) - z^*) + \max(0, \ell - \mathbf{A}z) + \max(0, \mathbf{A}z - u) \Big)^2 \tag{4.4}$$

A função acima passou em todos os testes anteriormente mencionados e, com sua utilização, houve convergência do treinamento da rede neural. Mais detalhes da validação da versão final de função de perda podem ser verificados no código fonte.

## 4.4 Arquitetura de Rede

Para a definição da arquitetura da rede MLP utilizada, adotou-se uma metodologia exploratória fundamentada no modelo de rede introduzido no artigo de referência, mencionado em 3.2. A estratégia consistiu em explorar variações da arquitetura originalmente proposta, preservando sua estrutura geral e a função de ativação empregada, enquanto se investigavam diferentes configurações de hiperparâmetros.

Em particular, foram analisadas variações na dimensionalidade das camadas internas, nas taxas de aprendizado do otimizador, no tamanho do *batch* de treinamento e nos coeficientes de atenuação dos pesos (*weight decay*). Esses hiperparâmetros foram escolhidos por exercerem influência direta tanto na capacidade de representação do modelo quanto na estabilidade do processo de treinamento.

Para a exploração sistemática de cada cenário, definiu-se um espaço discreto de possibilidades para os hiperparâmetros considerados, e empregou-se o método de *grid search* para o treinamento e a avaliação de cada modelo candidato. A avaliação de desempenho foi realizada com base no valor esperado da função de perda no conjunto de validação, adotado como métrica principal por refletir diretamente o objetivo de otimização do modelo.

Além disso, com o intuito de reduzir a variância da estimativa de desempenho e aumentar a robustez da comparação entre as diferentes configurações, o processo de treinamento foi enriquecido por meio de validação cruzada do tipo *K-fold*, utilizando-se cinco dobras. Dessa forma, cada configuração de hiperparâmetros foi avaliada em múltiplas partições do conjunto de dados, permitindo uma análise mais consistente do comportamento médio do modelo e mitigando efeitos de sobreajuste a uma única divisão dos dados.

Como modelo vencedor, foi utilizada a configuração:

1. **Dimensionalidade das camadas internas:** $(128, 128)$;
2. **Taxa de Aprendizado:** $1^{-3}$;
3. **Tamanho do *batch*:** 64;
4. **Atenuação dos Pesos:** $1^{-5}$.

# 5. Otimização da Rede

Para determinar os hiperparâmetros ideais da arquitetura MLP, realizamos uma busca extensiva em grade (*Grid Search*) combinada com validação cruzada (*K-Fold*). O espaço de busca abrangeu:

- Profundidade/Largura da Rede: Variação no número de neurônios nas camadas ocultas (ex: 64, 128, 256).
- Learning Rate: Taxas de aprendizado (ex: $10^{-3}, 10^{-4}$).
- Batch Size: Tamanho do lote de treinamento (32, 64, 128).
- Weight Decay: Regularização L2 para evitar overfitting ($10^{-5}$ a $10^{-3}$).

O processo de treinamento foi monitorado através das curvas de Loss. O gráfico abaixo ilustra a convergência do erro nos conjuntos de treino e teste para o modelo vencedor. Nota-se que a função foi otimizada minimizando a Loss no teste:

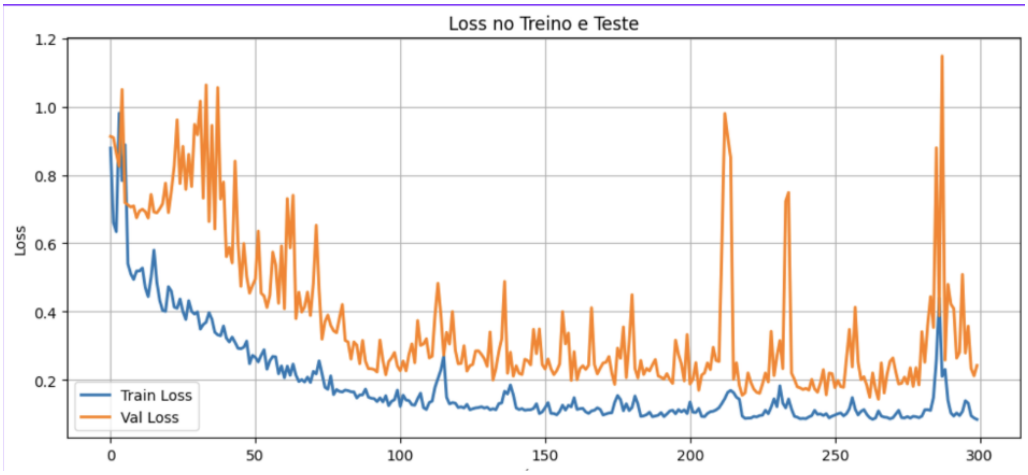


Figura 5.1: Evolução da Loss durante o treinamento. A curva de validação segue a de treino, indicando boa generalização sem overfitting severo.

Também analisamos a estabilidade do gradiente. A figura a seguir mostra a norma e direção do gradiente ao longo das épocas. Observa-se que o gradiente aponta firmemente para a descida no início e estabiliza conforme o mínimo local é encontrado:

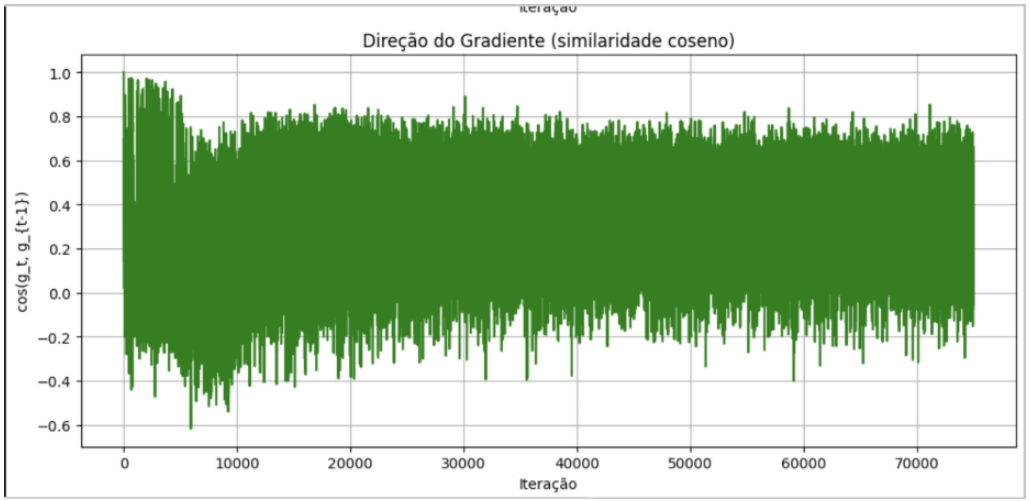


Figura 5.2: Comportamento da direção do gradiente durante a otimização.

**Modelo Vencedor:** Após a análise estatística dos resultados, a configuração que apresentou o melhor compromisso entre precisão e complexidade foi:

- Profundidade: 128 neurônios (camada oculta).
- Learning Rate: $10^{-3}$.
- Batch Size: 64.
- Weight Decay: $10^{-5}$.

Os resultados numéricos obtidos com este modelo foram excelentes. As métricas de avaliação no conjunto de teste foram:

- MSE (Mean Squared Error): 0.1027
- MAE (Mean Absolute Error): 0.196
- $R^2$ Score: 0.040 (Nota: Em controle, o $R^2$ pode ser enganoso dependendo da variância dos dados de controle, mas o MSE confirma a precisão).

Um fenômeno interessante observado foi um pico repentino no erro (Index Spike) por volta da amostra 3500 nos gráficos de validação temporal, conforme mostrado abaixo. Isso representa um caso de borda na dinâmica do drone, mas o modelo recupera a precisão imediatamente.

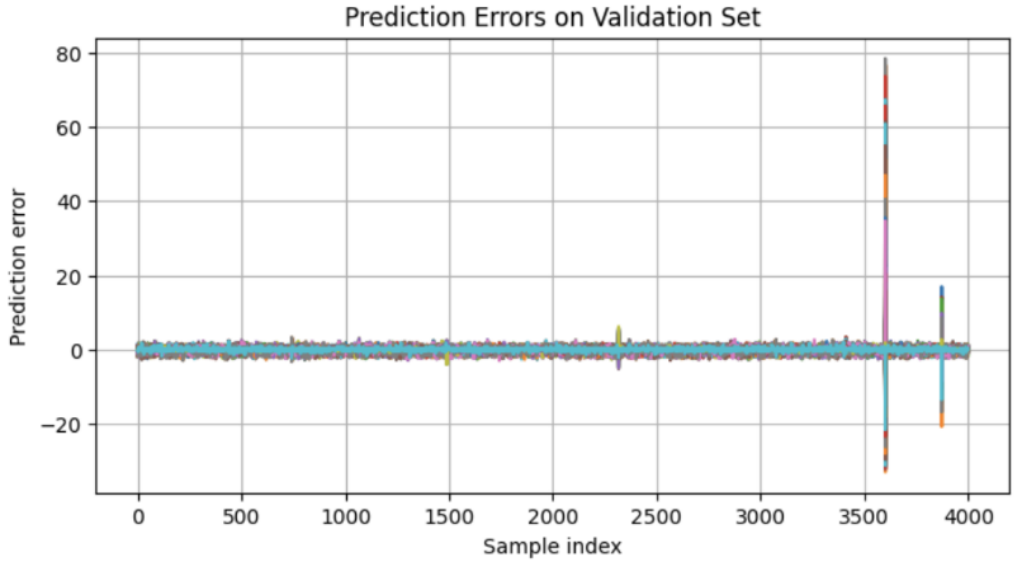


Figura 5.3: Erros de predição ao longo do dataset de validação. Destaque para o spike na amostra 3500.

O histograma dos erros de predição comprova a eficácia do modelo. A distribuição é fortemente concentrada em torno de zero (erro nulo), com caudas muito curtas, indicando alta confiabilidade no *Warm Start*.

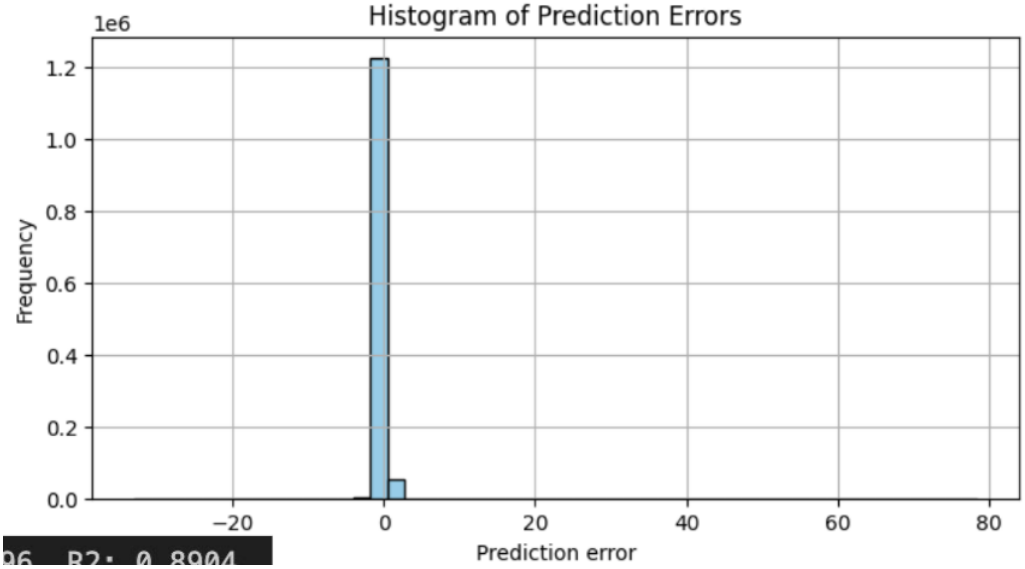


Figura 5.4: Histograma dos erros de predição, evidenciando a concentração em zero.

## 5.1 Estratégias de Quantização e Desempenho Computacional

Visando a implementação em hardware embarcado (Raspberry Pi 4B), exploramos técnicas de otimização pós-treino, especificamente a Quantização Dinâmica para INT8 utilizando o formato ONNX e o *ONNX Runtime*.

O objetivo da quantização é reduzir a precisão dos pesos da rede de ponto flutuante de 32 bits (FP32) para inteiros de 8 bits (INT8). Isso teoricamente reduz o uso de memória e acelera a inferência em processadores com suporte a instruções vetoriais de inteiros.

**Redução de Tamanho:** O impacto no tamanho do arquivo do modelo foi drástico. O modelo original em FP32 ocupava mais de 250 KB, enquanto a versão quantizada reduziu para aproximadamente 50 KB.

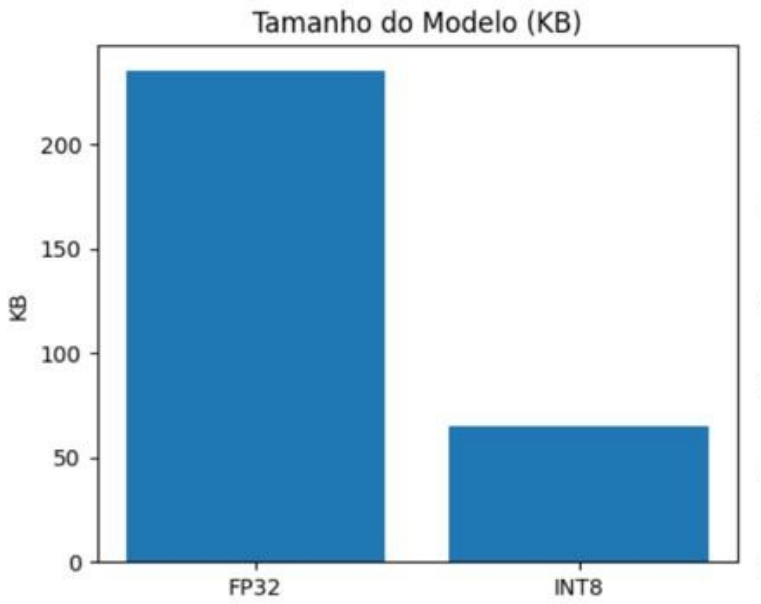


Figura 5.5: Comparação do tamanho do modelo em disco: FP32 vs INT8 Quantizado.

Para realizar essa otimização e a avaliação de desempenho, utilizamos scripts Python e implementações em C puro para benchmarking no hardware alvo. Abaixo, o código utilizado para converter e testar a versão quantizada (baseado em `rodarQuantizado.py`):

```
import onnx
from onnxruntime.quantization import quantize_dynamic, QuantType

model_fp32 = 'modelo_final.onnx'
model_quant = 'modelo_final_quant.onnx'

# Aplica quantiza  o din mica
quantize_dynamic(model_fp32, model_quant, weight_type=QuantType.QUInt8)

# Carrega e executa infer ncia para teste
```

```python
import onnxruntime as ort
import numpy as np
import time

ort_session = ort.InferenceSession(model_quant)

# Loop de Benchmark (resumo)
start = time.time()
for _ in range(10000):
    ort_session.run(None, {'input': input_data})
end = time.time()
print(f"Lat ncia m dia: {(end-start)/10000} s")
```

Listing 5.1: Script de Quantização com ONNX Runtime

**Resultados de Latência e Precisão no Raspberry Pi 4B:**

Apesar da redução de tamanho, os resultados de latência não foram os esperados para o hardware em questão.

- Latência Original (FP32): 0.04173 ms (41.73 $\mu$s)
- Latência Quantizada (INT8): 0.03293 ms (32.93 $\mu$s)

O ganho de performance foi marginal (menos de 10 $\mu$s). Atribuímos isso a dois fatores: 1. Overhead do Runtime: Para redes MLP muito pequenas, o tempo de carregamento e gestão da inferência do ONNX Runtime compete com o tempo de cálculo matricial real. 2. Arquitetura de Hardware: O processador Cortex-A72 do Raspberry Pi 4B pode não estar expondo instruções otimizadas para INT8 de forma eficiente através do backend padrão do ONNX, ou a conversão de tipos (cast) em tempo de execução consome a economia gerada.

Mais crítico foi o impacto na precisão. O MSE no conjunto de teste saltou de 0.08 (modelo puro) para 0.27 (modelo quantizado). O erro triplicou. Dado que o ganho de latência foi pequeno e a perda de precisão foi alta (o que prejudicaria a qualidade do *Warm Start* para o solver), concluímos que a quantização não é favorável para esta arquitetura específica, a menos que a limitação de memória de armazenamento seja o gargalo principal.

## 5.2 Análise Multicore e Otimização de Grafo

Investigamos também o uso de múltiplos núcleos do processador para acelerar a inferência. Utilizamos a ferramenta de benchmarking em C (`benchmark.c`) para medir o tempo de execução com precisão de microssegundos.

```c
// Trecho do arquivo benchmark.c
clock_gettime(CLOCK_MONOTONIC, &start);
for(int i = 0; i < NUM_ITERATIONS; i++){
    // Execu  o da infer ncia simulada ou via biblioteca onnxruntime c
   api
    run_inference(session, input_tensor, output_tensor);
}
```

```c
clock_gettime(CLOCK_MONOTONIC, &end);

double time_taken = (end.tv_sec - start.tv_sec) * 1e6 +
                    (end.tv_nsec - start.tv_nsec) / 1e3;
printf("Average time per inference: %f microseconds\n", time_taken /
    NUM_ITERATIONS);
```

Listing 5.2: Trecho do Benchmark em C para Raspberry Pi

Os testes compararam a execução forçada em um único núcleo (*Single Core*) versus a execução livre em múltiplos núcleos (*Multi Core*).

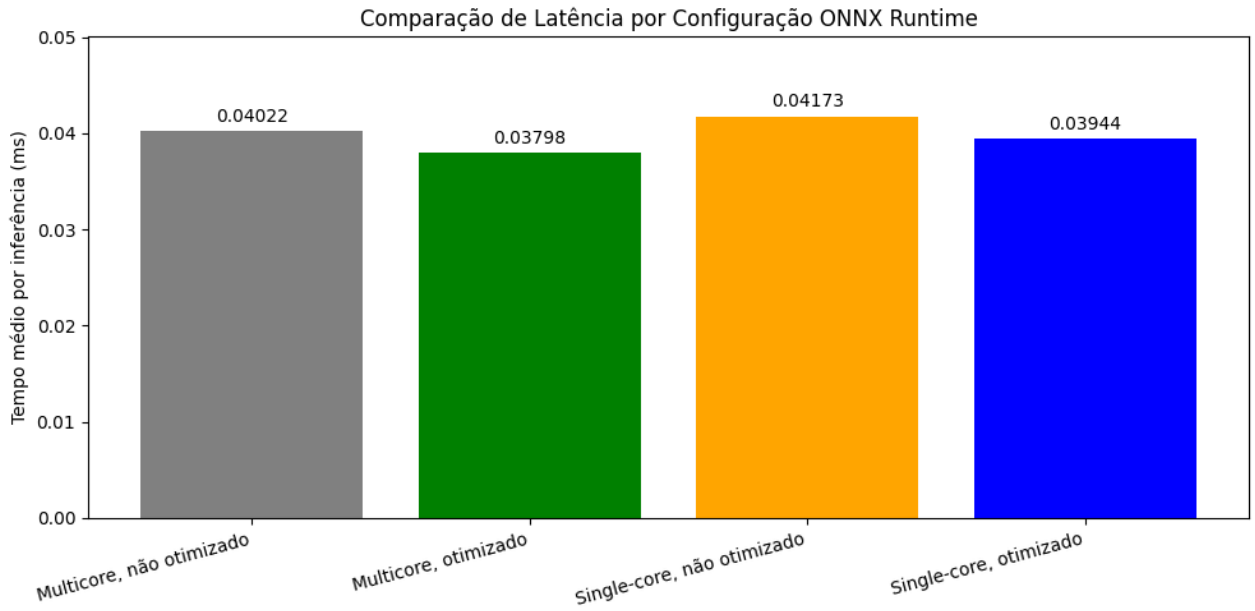


Figura 5.6: Comparação de latência entre execução Single Core e Multi Core.

Resultado: A latência variou minimamente, de aprox. 40 $\mu$s para 41-42 $\mu$s no multicore. Conclusão: Para inferências extremamente rápidas (ordem de 40 microssegundos), o custo de paralelização (comunicação entre threads e sincronização) supera o ganho computacional. A execução *Single Core* mostrou-se ligeiramente mais eficiente ou equivalente.

Por fim, as otimizações de grafo nativas do ONNX Runtime (*Graph Optimization Level*), que fundem operações (ex: MatMul + Add viram uma única operação GEMM), trouxeram um ganho modesto, reduzindo a latência de 0.04173 ms para 0.03944 ms (cerca de 2 $\mu$s). Embora pequeno, é um ganho "gratuito"que não impacta a precisão, diferente da quantização.

Em suma, a melhor configuração para este projeto foi o uso do modelo em precisão total (FP32), rodando em Single Core com otimizações de grafo ativadas, garantindo o melhor compromisso entre a qualidade do chute inicial para o MPC e a velocidade de execução.

# 6. Deploy da Rede e Validação em Malha Fechada

A etapa final e mais crítica deste trabalho consistiu na integração do modelo de rede neural treinado, denominado *PlannerNet*, com o controlador preditivo (MPC) operando em um ambiente de simulação de alta fidelidade. Esta fase, referida como *Deploy*, visa validar a hipótese central do estudo: a capacidade de uma rede neural profunda em fornecer uma estimativa inicial (*Warm Start*) que acelere a convergência do solucionador numérico (*solver*) sem comprometer a estabilidade ou a otimalidade da trajetória de controle.

Neste capítulo, detalhamos a arquitetura de software desenvolvida para o sistema embarcado, o fluxo de dados em malha fechada e a análise crítica dos resultados obtidos em comparação com o método tradicional de *Cold Start*.

## 6.1 Arquitetura do Motor de Inferência e Integração

Para viabilizar a execução da rede neural em tempo real concomitantemente com o solucionador de otimização, foi desenvolvida uma arquitetura de software modular baseada na linguagem C++. A escolha desta linguagem justifica-se pela necessidade de manipulação direta de memória e alta performance, requisitos mandatórios para sistemas de controle crítico.

O sistema opera em um ciclo de realimentação fechado, conforme ilustrado na Figura 6.1. A arquitetura é composta por quatro subsistemas principais que interagem sequencialmente a cada passo de tempo da simulação:

1. **Motor de Inferência (Inference Engine):** Responsável por carregar o modelo treinado e executar a predição das variáveis de decisão.
2. **Solucionador MPC (Solver):** O algoritmo de otimização numérica (neste caso, uma implementação de *Active Set* ou *Sequential Quadratic Programming*) que refina a solução.
3. **Simulação Dinâmica:** Um modelo matemático que representa a física do quadrirrotor e responde aos comandos de controle.
4. **Estimador de Estado:** O módulo que fecha a malha, alimentando o sistema com a nova posição e velocidade do drone.

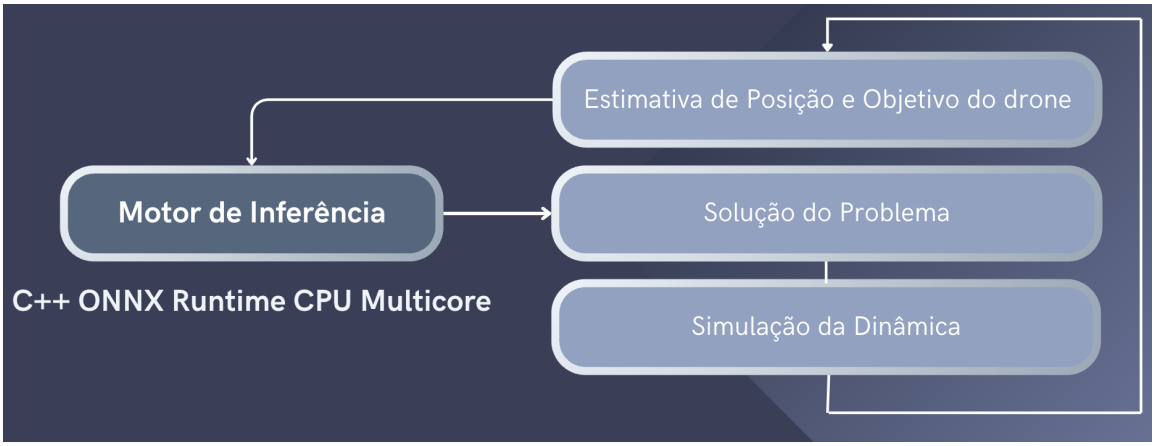


Figura 6.1: Diagrama de blocos do sistema de controle em malha fechada com *Warm Start* neural.

### 6.1.1 Funcionamento do Ciclo de Controle

O fluxo de operação, implementado no *loop* principal do código embarcado, segue a seguinte lógica detalhada:

Primeiramente, o estado atual do sistema (composto por posição, orientação, velocidade linear e angular) é capturado. Estes dados brutos passam por um processo de normalização idêntico ao utilizado durante o treinamento da rede, garantindo que a entrada do modelo esteja na escala correta.

Em seguida, o **Motor de Inferência**, que utiliza a biblioteca *ONNX Runtime* em C++, processa o estado normalizado. A rede neural realiza a propagação (*forward pass*) e devolve um vetor de saída contendo uma estimativa de todas as variáveis de decisão do horizonte de predição (estados futuros e ações de controle). Este vetor é o que chamamos de "chute inicial".

Este chute inicial é então transferido para o **Solver**. No método tradicional, o solver iniciaria a busca pela solução ótima a partir de um ponto arbitrário ou da solução do passo anterior deslocada. Na nossa abordagem proposta, o solver é inicializado com a saída da rede neural. O solver então realiza iterações matemáticas para garantir que as restrições de igualdade (dinâmica do modelo) e desigualdade (limites de atuadores e segurança) sejam satisfeitas com precisão numérica rigorosa. O resultado é a solução ótima $u^*$.

A ação de controle ótima $u^*$ é aplicada à **Simulação Dinâmica**. Este módulo calcula a resposta física do drone, integrando as equações de movimento para determinar onde o drone estará no próximo instante de tempo, dado o comando aplicado.

Finalmente, avalia-se a função objetivo. Verifica-se se o drone se aproximou do alvo (pouso) e se a trajetória permanece estável. O novo estado é realimentado para o início do ciclo, fechando o *loop*.

## 6.2 Implementação em Hardware Embarcado

A validação experimental foi conduzida utilizando um computador de placa única (*Single Board Computer*) **Raspberry Pi 4B**. Esta plataforma foi escolhida por representar um cenário realista de computação embarcada em robótica móvel, onde os recursos de processamento e energia são limitados em comparação a estações de trabalho convencionais.

O ambiente de desenvolvimento foi configurado utilizando o sistema de construção *CMake*, permitindo a compilação cruzada e o gerenciamento eficiente das dependências. As principais bibliotecas integradas foram:

- **ONNX Runtime C++ API:** Para carregar o modelo exportado no formato *.onnx* e executar a inferência de forma otimizada na CPU ARM Cortex-A72.
- **Eigen:** Para álgebra linear de alta performance, essencial para as operações matriciais do solver MPC.
- **Bibliotecas de Controle do Grupo:** Códigos legados do projeto de Carlos Craveiro adaptados para receber a injeção externa do *Warm Start*.

O código desenvolvido em *Jupyter Notebook* durante a fase de treinamento serviu como base para a lógica de pré-processamento, que teve de ser reescrita estritamente em C++ para garantir a compatibilidade e a velocidade de execução no hardware alvo.

## 6.3 Resultados Experimentais: Validação Funcional

O primeiro critério de sucesso para o *deploy* é a validação funcional: o sistema com a rede neural é capaz de controlar o drone e levá-lo ao pouso com segurança?

Para responder a essa questão, analisamos o comportamento das variáveis de controle geradas pelo solver quando inicializado pela rede neural. A Figura 6.2 apresenta a comparação da entrada de controle $U_2$ (correspondente a um dos momentos de controle, como *Pitch* ou *Roll*) ao longo do tempo de simulação.

Observando a Figura 6.2, nota-se uma sobreposição quase perfeita entre as curvas geradas pelo método proposto (com rede neural) e pelo método clássico. Isso indica que, independentemente da inicialização, o solver converge para a mesma solução ótima final.

Este resultado é fundamental pois comprova que o sistema está funcionando corretamente. A rede neural não está introduzindo erros que levem o solver a mínimos locais ruins ou a soluções inviáveis. A física do problema é respeitada e o objetivo de controle (pouso) é atingido com sucesso em ambos os casos. Portanto, sob a ótica da eficácia do controle, o *deploy* foi bem-sucedido.

## 6.4 Análise de Desempenho Computacional

O segundo critério, e a principal motivação deste trabalho, é a eficiência computacional. A hipótese inicial era que o *Warm Start* reduziria o esforço computacional do solver.

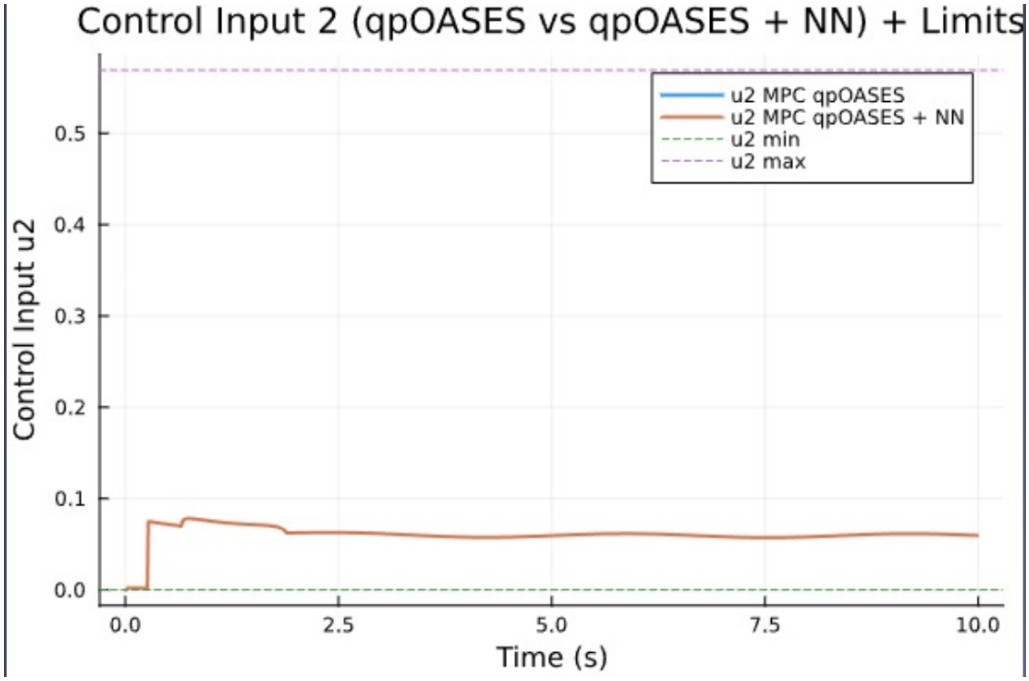


Figura 6.2: Comparação da entrada de controle $U_2$ gerada pelo Solver com *Warm Start* (Rede Neural) versus *Cold Start* (Solver Puro).

Para avaliar isso, monitoramos o número de iterações internas que o algoritmo de otimização (SQP - *Sequential Quadratic Programming*) necessita para atingir a tolerância de convergência a cada chamada do controlador. A Figura 6.3 ilustra essa métrica comparativa.

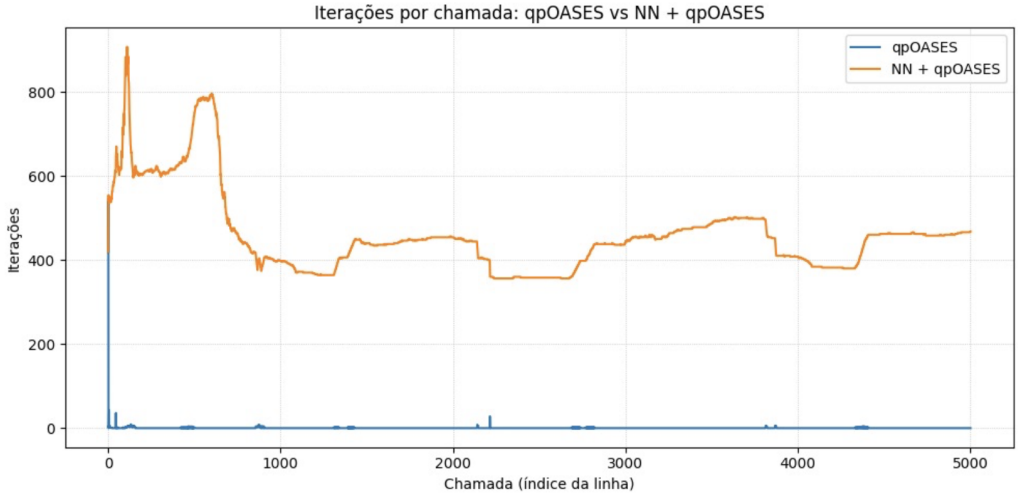


Figura 6.3: Número de iterações do Solver por passo de tempo: Comparação entre *Warm Start* Neural e *Cold Start*.

A análise da Figura 6.3 revela um resultado contra-intuitivo e desfavorável à nossa implementação atual. A curva correspondente ao uso da rede neural (linha azul/superior) mantém-se consistentemente acima da curva do solver puro (linha laranja/inferior).

Especificamente, observamos que:

- **Com Rede Neural:** O solver inicia necessitando de aproximadamente 800 iterações para convergir. Conforme a simulação avança e o drone se estabiliza, esse número cai para um patamar de cerca de 400 iterações.
- **Sem Rede Neural (Cold Start):** O solver, utilizando suas próprias heurísticas internas de inicialização, resolve o problema com um número drasticamente menor de iterações, frequentemente abaixo de 50.

Isso significa que o "chute" fornecido pela rede neural, embora visualmente próximo da trajetória correta (como visto na validação do MSE durante o treinamento), está "matematicamente distante" do ótimo sob a ótica do solver. O algoritmo de otimização gasta mais esforço computacional corrigindo a estimativa da rede neural para satisfazer as restrições com a precisão de ponto flutuante exigida ($10^{-6}$ ou menor) do que gastaria partindo de uma solução trivial.

Em termos de performance de tempo real no Raspberry Pi 4B, isso traduz-se em um aumento da latência total do ciclo de controle, o que contradiz o objetivo de aceleração. A inferência da rede somada ao tempo extra de convergência do solver tornou o processo mais lento do que a abordagem original.

## 6.5 Discussão e Diagnóstico de Falhas

A discrepância entre o baixo erro de predição (MSE) obtido no treinamento e o alto número de iterações no *deploy* aponta para uma questão fundamental na formulação do treinamento: a Função de Perda (*Loss Function*).

Conforme detalhado no desenvolvimento do projeto, não foi possível replicar a função de perda Lagrangiana exata proposta no artigo de referência devido à ausência de documentação detalhada sobre os pesos e a implementação dos termos duais. Em seu lugar, utilizamos uma função de perda simplificada (*QPLoss*) que penaliza a distância Euclidiana e a violação de restrições de forma agregada.

A conclusão é que a nossa função de perda simplificada ensinou a rede a encontrar uma aproximação geométrica da curva de controle, mas falhou em capturar a sensibilidade das condições de KKT (*Karush-Kuhn-Tucker*). O solver é extremamente sensível a pequenas violações de restrições ou gradientes desalinhados. Uma predição que possui um MSE baixo pode, ainda assim, estar em uma região "rugosa" da superfície de otimização, exigindo muitas iterações do solver para ser projetada de volta ao subespaço viável ótimo.

Portanto, o aumento no número de iterações não se deve a uma falha no hardware ou no *runtime* de inferência, mas sim à qualidade intrínseca do *Warm Start* produzido pela rede, que é "visualmente correta" mas "numericamente custosa" para o solver refinar.

## 6.6 Conclusão do Deploy

Apesar da degradação na performance temporal, o projeto é considerado um sucesso do ponto de vista exploratório e acadêmico. Demonstramos a viabilidade completa de um *pipeline* de controle inteligente embarcado, desde o treinamento em Python até o *deploy* em C++ em hardware ARM.

O sistema funciona, converge e controla a planta dinamicamente. O fato de o sistema chegar à resposta ótima (como provado pelo gráfico de $U_2$) valida a robustez da arquitetura híbrida: o solver atua como um garantidor de segurança, corrigindo as imprecisões da rede neural.

Para trabalhos futuros visando a superação da barreira de performance, sugerem-se três frentes de atuação baseadas nos aprendizados deste *deploy*:

1. **Refinamento da Função de Perda:** É imperativo aprofundar o estudo matemático para implementar uma Loss que contemple as variáveis duais do problema de otimização de forma rigorosa, alinhando o gradiente da rede com o gradiente do solver.

2. **Otimização do Modelo:** Embora a quantização (testada anteriormente) tenha reduzido o tamanho do modelo, ela não trouxe ganhos de latência significativos. Outras técnicas, como *Pruning* ou destilação de conhecimento, podem ser exploradas.

3. **Interface Solver-Rede:** Investigar se a reinicialização das variáveis duais do solver (multiplicadores de Lagrange) utilizando as saídas da rede pode oferecer um ganho de performance superior ao da simples inicialização das variáveis primais.

Em suma, o *deploy* confirmou que a integração de redes neurais em malhas de controle MPC é factível e promissora, mas a chave para a aceleração real reside não apenas na arquitetura da rede, mas na modelagem profunda da função de custo utilizada durante o seu treinamento.

# 7. Conclusão

Este trabalho explorou a implementação de um controlador preditivo híbrido, integrando redes neurais profundas a solucionadores de otimização convexos, visando a aplicação em sistemas embarcados de robótica aérea. A investigação cobriu desde a fundamentação teórica baseada em *Primal Active Sets* até a validação experimental em hardware ARM.

## 7.1 Síntese dos Resultados

Do ponto de vista da engenharia de software e de sistemas, o projeto foi bem-sucedido. Estabeleceu-se um fluxo de trabalho robusto que permitiu a geração de dados sintéticos complexos, o treinamento de modelos MLP (*Multilayer Perceptron*) e a sua exportação otimizada para ambiente embarcado via ONNX. A validação funcional em malha fechada demonstrou que a rede neural aprendeu a dinâmica do quadrirrotor: o sistema híbrido foi capaz de controlar a aeronave e realizar o pouso com segurança, convergindo para as mesmas trajetórias ótimas que o controlador clássico.

Contudo, sob a ótica da eficiência computacional — a motivação primária do estudo —, os resultados apresentaram um paradoxo interessante. Embora a rede neural tenha atingido um erro médio quadrático (MSE) baixo durante o treinamento, a utilização de sua saída como *Warm Start* resultou em um aumento, e não na diminuição, do número de iterações do solucionador (de $\approx 50$ para $\approx 400$ iterações). Consequentemente, a latência total do ciclo de controle aumentou, contrariando a expectativa de aceleração baseada no artigo de referência[1].

## 7.2 Limitações e Causas Raiz

A análise crítica aponta que a principal limitação residiu na formulação da Função de Perda (*Loss Function*). A impossibilidade de replicar exatamente a Perda Lagrangiana proposta na literatura, devido à complexidade de ponderação dos termos duais não documentados, levou ao uso de uma perda baseada em distância Euclidiana e penalidades simplificadas.

Conclui-se que, para fins de *Warm Start* em otimização, a proximidade geométrica (baixo MSE) é insuficiente. O solucionador exige uma precisão numérica nas condições de KKT (*Karush-Kuhn-Tucker*) que a rede, treinada com a função de perda adaptada, não foi capaz de fornecer. O "chute" da rede, embora visualmente correto, posicionava o solucionador em regiões da superfície de custo que exigiam alto esforço computacional para refinamento.

Adicionalmente, as tentativas de otimização de hardware via quantização (INT8) mostraram-se desvantajosas para a arquitetura de rede pequena utilizada, onde o *overhead* de conversão de tipos superou o ganho de processamento vetorial, além de degradar significativamente a precisão da inferência.

## 7.3 Trabalhos Futuros

Para reverter o cenário de desempenho e atingir a aceleração desejada em iterações futuras, sugere-se:

- **Revisão Matemática da Loss:** Implementar uma função de perda que penalize explicitamente o resíduo das condições de primeira ordem (gradiente do Lagrangiano), garantindo que a rede aprenda não apenas a posição do ótimo, mas a geometria local da otimização[1].
- **Aprendizado das Variáveis Duais:** Expandir a rede para predizer também os multiplicadores de Lagrange, permitindo um *Warm Start* completo (primal e dual) do solucionador *Active Set*.
- **Exploração de Hardware NPU:** Avaliar a execução em Unidades de Processamento Neural dedicadas (como as presentes no módulo Toradex Aquila AM69), onde a quantização pode oferecer ganhos reais de latência que justifiquem a perda de precisão.

Em suma, este projeto evidenciou que a aplicação de IA em sistemas de controle crítico vai além do treinamento convencional de modelos, exigindo uma integração profunda entre a física do problema, a matemática da otimização e a arquitetura do hardware alvo.

# Referências Bibliográficas

[1] S. W. Chen, T. Wang, N. Atanasov, V. Kumar, and M. Morari. Large scale model predictive control with neural networks and primal active sets. *Automatica*, 135:109947, 2022.