UNIVERSIDADE FEDERAL DA BAHIA

ESCOLA POLITÉCNICA

DEPARTAMENTO DE ENGENHARIA MECÂNICA

VINÍCIUS MARQUES MATOS

ANÁLISE DE SISTEMA DE INFERÊNCIA ADAPTATIVO NEURO-FUZZY – ANFIS

Salvador

2021

VINÍCIUS MARQUES MATOS

ANÁLISE DE SISTEMA DE INFERÊNCIA ADAPTATIVO NEURO-FUZZY - ANFIS

Trabalho de Conclusão de Curso apresentado à Escola Politécnica da Universidade Federal da Bahia, como requisito parcial para a obtenção do título de Bacharel em Engenharia Mecânica.

Orientador: Dr. Marcelo Costa Tanaka.

Salvador

2021

TERMO DE APROVAÇÃO

VINÍCIUS MARQUES MATOS

ANÁLISE DE SISTEMA DE INFERÊNCIA ADAPTATIVO NEURO-FUZZY - ANFIS

Trabalho de Conclusão de Curso apresentado à Escola Politécnica da Universidade Federal da Bahia, como requisito parcial para a obtenção do título de Bacharel em Engenharia Mecânica.

Banca examinadora do Trabalho apresentado e aprovado em 12/06/2021

_____

Marcelo Costa Tanaka, D.Sc.

_____

Bruno da Cunha Diniz, D.Sc.

_____

Josiane Maria de Macedo Fernandes, D.Sc.

AGRADECIMENTOS

Ao Prof. Marcelo costa Tanaka, pela oportunidade que me concedeu em ser meu orientador para este trabalho.

Aos meus pais que suportou os melhores e piores momentos, sempre me apoiando com amor e paciência. À minha avó, que cuidou de mim e me ensinou o caminho de Deus. À minha família que me apoiou de longe com atenção e carinho. À minha noiva, que me incentivou e orou pelo meu sucesso.

Aos meus colegas de graduação, mas em especial a Lenon Cerqueira e a Maurício Gandarela, sem os quais as tardes na faculdade não seriam as mesmas e me incentivaram a chegar até aqui.

“Ainda que eu andasse pelo vale da sombra da morte, não temerei mal algum, porque tu estás comigo; tua vara e teu cajado me consolam.”

SALMO 23

RESUMO

A linguagem *fuzzy* é amplamente utilizada para construção de modelos de controle, previsão, mapeamento de funções, entre outros. Entre suas principais vantagens, destaca-se o fácil entendimento da contribuição de seus componentes em seu resultado. Apesar disso, devido a sua forma de construção altamente dependente do especialista humano e a não implementação nativa de algoritmos de aprendizagem, o processo de construção de um sistema *fuzzy* necessita de diversas iterações manuais para se obter um resultado satisfatório. O sistema de inferência adaptativo *neuro-fuzzy* busca solucionar este fato realizando a implementação de algoritmos de aprendizagem através do conceito de redes adaptativas, retirando parte da responsabilidade do especialista humano na definição do sistema. Desta forma, neste trabalho é realizado uma análise deste sistema, utilizando um algoritmo de aprendizagem baseado nos métodos de gradiente reverso e mínimos quadrados. A partir da estrutura descrita, obtém-se diversas iterações em um sistema seno cardinal de duas variáveis buscando-se isolar alguns dos parâmetros de construção do sistema e analisar o impacto deles na performance de erro do sistema. Obtém-se resultados relevantes quanto à generalização do sistema com base nas variáveis, sua aprendizagem e como os parâmetros influenciam no resultado do sistema, além de considerações a escolhas de projeto que podem influenciar de forma positiva no resultado do sistema.

Palavras-chave: *Lógica fuzzy; Sistema de inferência adaptativo neuro-fuzzy; Gradiente reverso; Mínimos quadrados.*

ABSTRACT

The fuzzy logic is widely used on the construction of control models, forecasting models, function mapping and other applications. One of the many advantages of the fuzzy system is the intuitiveness of the contribution each component provides to the result. Despite that, given the high dependency on the human expert and no native learning algorithm, the construction of a fuzzy system demands a high number of manual iterations to provide a satisfactory result. The adaptative neuro-fuzzy inference system try to solve this issue by implementing learning algorithms through the concept of adaptive networks, removing part of the responsibility of the human expert on the system definition. This way, this paper performs an analysis of this system utilizing a learning algorithm based on the gradient descendance and the least square error algorithms. Based on this structure, a few iterations are performed on a sine cardinal function with two independent variables, trying to isolate some of the system construction parameters and analyze the impact of each on the system performance. Relevant results regarding the generalization of the system based on its variables, its learning capability and system parameters are obtained, in addition of considerations regarding project choices that may influence the system result positively.

Key words: *Fuzzy logic; Adaptative neuro-fuzzy inference system; Gradient descendance; Least square error.*

LISTA DE SIGLAS

PID	Proporcional Integral Derivativo
PI	Proporcional Integral
P	Proporcional
RNA	Redes Neurais Artificiais
ANFIS	Sistema de Inferência *Fuzzy* Baseado em Redes Adaptativas
TKS	Takagi, Sugeno e Kang
RMSE	Raiz do erro quadrático médio

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

As funções de controle estão presentes em boa parte dos processos necessários para o dia a dia de todos. Do controle de temperatura do centro de processamento de um computador até o controle da movimentação de braços mecânicos para processos industriais automatizados, essas funções se fazem essenciais para produção de boa parte dos produtos que são utilizados hoje.

A maior parte destes processos são relativamente simples de serem simulados e controlados, não necessitando de grandes esforços para atingir resultados dentro de uma margem esperada. Normalmente um controlador Proporcional Integral Derivativo (PID), ou alguma de suas variações, como controladores Proporcionais (P) ou Proporcionais Integrativos (PI), são os mais utilizados: “Mais de 90% dos controladores em operação hoje são controladores PID (ou alguma forma de controlador PID como controladores P ou PI).” (PASSINO e YURKOVICH, 1998, p. 7, tradução nossa).

Entretanto, quando se trata de problemas mais complexos faz-se necessário uma abordagem diferente para a construção do modelo a ser analisado. Esses modelos são cada vez mais necessários para atingir a expectativa de eficiência e qualidade exigidos pela indústria e pela população na utilização de recursos como energia e insumos naturais.

Devido a isto, sistemas não lineares utilizando tecnologias de inteligência artificial como Redes Neurais Artificiais (RNA) e sistemas de lógica difusa (*fuzzy logic*), comumente utilizados em problemas complexos, tem sido amplamente explorados para se alcançar níveis aceitáveis de performances.

Dentre estes, a RNA se destaca pela capacidade inerente de aprendizagem, necessária para o funcionamento desta, que adapta os parâmetros da rede de forma a tender seus resultados a um erro mínimo.

Entretanto, para alcançar tal feito, as redes neurais tendem a necessitar de muitos dados para seu treinamento, que por sua vez impacta na performance e na memória necessária para seu treinamento.

Por outro lado, os sistemas que utilizam a lógica difusa (sistemas *fuzzy*) normalmente não possuem qualquer forma de aprendizado automatizado em sua construção, sendo necessário para seu funcionamento informações prévias do sistema a ser modelado, denominado de especialista humano, que transfere parte do conhecimento prévio adquirido sobre o problema para a inteligência artificial através de regras heurísticas.

Dada essa transferência de conhecimento, o método de funcionamento de um sistema *fuzzy* é muito mais simples de ser compreendido em suas etapas constituintes quando comparado a um sistema "caixa-preta", como uma RNA, em que não há uma definição intuitiva da contribuição de cada componente no resultado obtido.

Entretanto dois problemas típicos são comuns durante o desenvolvimento de sistemas *fuzzy*: a não existência de um especialista capaz de fornecer informações suficientes para modelagem do problema de forma satisfatória e a necessidade de ajuste dos parâmetros do sistema de forma manual e empírica. Tem-se ao final do processo um sistema que necessitou de diversas iterações manuais para se alcançar um resultado satisfatório.

> O problema com isso é que comumente existem parâmetros demais para ajustar [...] e normalmente não há uma conceção clara entre o objetivo do design, [...] um racional e um método que deva ser usado para ajustar estes parâmetros. (PASSINO e YURKOVICH, 1998, p. 83, tradução nossa).

Devido a isto, fez-se necessário a construção de um modelo *fuzzy* que utilize os conceitos de aprendizagem de máquina presentes nas redes neurais através da implementação de algoritmos de aprendizagem, como o gradiente reverso, algoritmos genéticos e semelhantes.

O crescente número de trabalhos relacionados a algoritmos de lógica difusa adaptativos demonstra o interesse da comunidade científica no assunto. Os mais comuns são criações de modelos de previsão, como: previsões atmosféricas (RAJKUMAR *et al*., 2015); previsões das condições ambientais de solo (MEHDIZADEH *et al*., 2017) e água (XIE *et al*., 2016); previsão do valor de estoque do mercado automobilístico (SORAYAEI *et al*., 2012). Também é comum a criação de modelos de controle, como sistemas de rastreamento de pontos de potência máxima na geração de energia eólica (MEHARRAR *et al*., 2010) e maremotriz (SHEN *et al*., 2020), além de estudos teóricos como aplicação de sistemas *fuzzy* adaptativos

no plano complexo (SHOORANGIZ e MARHABAN, 2013), todos com altas taxas de sucesso na utilização da tecnologia. Vale ressaltar que apesar de existirem diversos métodos adaptativos de sistemas *fuzzy*, a maior parte dos trabalhos citados utiliza a metodologia de Sistema de Inferência *Fuzzy* Baseado em Redes Adaptativas (*Adaptive-Network-Based Fuzzy Inference System* – ANFIS, também comumente chamado de Sistema de Inferência Adaptativo Neuro-*Fuzzy* pela bibliografia) proposta por Jang (1993).

Com base nos sistemas e metodologias apresentados, este trabalho visa realizar uma análise aprofundada do modelo proposto por Jang (1993) para um sistema seno cardinal (*sinus cardinalis*) de duas variáveis independentes. O modelo consiste em um ANFIS com aprendizagem baseada nos algoritmos de mínimos quadrados e gradiente reverso.

1.1 Objetivo

Visando diminuir o número de iterações manuais para construção de sistemas *fuzzy*, este trabalho tem por objetivo analisar a performance do sistema adaptativo proposto por Jang (1993), comparando-o com um sistema *fuzzy* estático. Para tanto, se utiliza de um ANFIS com diversas variações de parâmetros em suas construções. Ainda como objetivos específicos, destacam-se:

- Expandir os resultados obtidos por Jang (1993) em termos de exemplos, condições iniciais e conjuntos de validação;
- Análise da contribuição dos parâmetros de construção básicos utilizados por Jang (1993);
- Análise dos parâmetros do método de aperfeiçoamento da propagação reversa (*backpropagation*), a propagação resiliente (*resilient-propagation*), conforme proposto por Jang (1993).

1.2 Organização do trabalho

Este trabalho é composto por seis capítulos, sendo este primeiro a apresentação da motivação inicial para realização do trabalho e a importância deste estudo para a comunidade científica. Neste capítulo são apresentados os principais aspectos das dificuldades presentes nos sistemas de lógica difusa atuais, comparando-o com as neurais artificiais.

No capítulo 2 é apresentado o conceito de lógica difusa, comparando-a à lógica matemática comum. Nele é descrito como a lógica difusa pode ser utilizada para criação de modelos de controle e previsão, por exemplo.

No capítulo 3 é apresentado a base para a construção de sistemas de inferência adaptativos neuro-*fuzzy,* demonstrando sua estrutura geral e a construção matemática de cada uma de suas camadas. Nele também é descrito o funcionamento dos algoritmos de aprendizagem utilizado deste trabalho.

No capítulo 4 é exposto da maior parte dos parâmetros de ajuste presentes no ANFIS, ressaltando aqueles que serão analisados no capítulo 5. Também é realizado a derivação do algoritmo de aprendizagem por gradiente reverso para a arquitetura proposta por Jang (1993).

No capítulo 5 é realizado a implementação do sistema apresentado nos capítulos 3 e 4 para mapeamento de uma função seno cardinal. Nele é apresentado resultados numéricos e aproximações da curva seno cardinal no espaço, visando-se isolar as variáveis destacadas no capítulo 4.

No capítulo 6 são apresentadas as considerações finais e as conclusões obtidas, bem como a proposta de desenvolvimento de novos trabalhos, visando expandir os resultados obtidos e analisar outras variáveis que foram tornados fixos neste trabalho.

No apêndice A é exposto o código base do ANFIS utilizado neste trabalho.

2 LÓGICA *FUZZY*

Conforme comenta Tanaka (2011), a lógica matemática comum é baseada nos conceitos básicos propostos por Aristóteles (384 – 322 a.C.), possuindo regras rígidas que devem ser seguidas de forma absoluta. Entre os princípios, destaca-se o princípio da não contradição, que propõe que uma proposição deve ser falsa ou verdadeira, não podendo ser ambas ao mesmo tempo:

Se e somente se o céu estiver limpo, **então** irei à praia.

O princípio do terceiro excluído também deve ser destacado, que propõe que uma proposição precisa ser necessariamente ou verdadeira ou falsa, não existindo uma terceira opção:

Ou o céu está limpo **ou** o céu não está.

Apesar de coerente, estes princípios não contemplam de forma absoluta a realidade, sendo insuficiente quando se trata de conceitos como “muito”, “pouco” e “talvez”, que por sua vez são amplamente utilizados no processo de decisão humano.

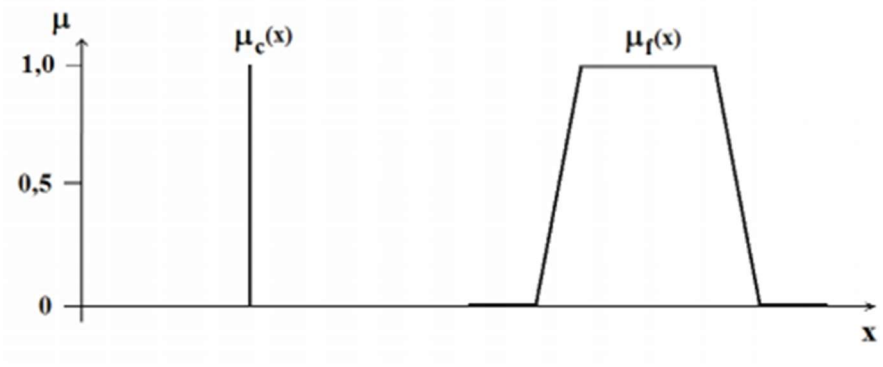
Neste sentido a lógica *fuzzy* propõe um conceito que não considera o princípio do terceiro excluído, passando a considerar não apenas “sim” e “não” mas avaliando um “grau de verdade” para a proposição. Desta forma, conceitos como “muito”, “provável” e “em torno de” podem ser alcançados. Por exemplo:

O céu está **um pouco** limpo.

Quando comparado ao sentido matemático de conjuntos, a forma clássica propõe que um elemento pertence ou não a um conjunto $A = \{x \mid x = 1\}$, enquanto a lógica *fuzzy* determina que um elemento possui um grau de pertencimento a um conjunto, definida através da função μ_a (função de pertinência), que classifica o elemento em um grau de pertinência dentro do intervalo [0,1], realizando uma transição suave entre “pertencer” e “não pertencer”.

A comparação entre os dois sistemas citados pode ser observada na Figura 2.1, em que μ_c é a função de pertinência do conjunto convencional e μ_f é a função de pertinência da lógica *fuzzy*:

Figura 2.1: Representação de um conjunto convencional e de um conjunto difuso



Fonte: Tanaka (2011, p. 19)

As funções de pertinência mais utilizadas nos sistemas *fuzzy* incluem funções trapezoidais, triangulares, a função gaussiana e algumas outras em forma de sino. Apesar de alterarem de forma significativa o comportamento e classificação dos valores de pertinência, não existe um método objetivo claro para a escolha destas: "[...] é importante notar que, em grande parte, a definição da função de pertinência é subjetiva. Isto é, definimos de uma forma numérica que faz sentido para nós, mas outro pode defini-la de forma diferente." (PASSINO e YURKOVICH, 1998, p. 34, tradução nossa).

2.1 Interseção de conjuntos *fuzzy* – operador lógico "e"

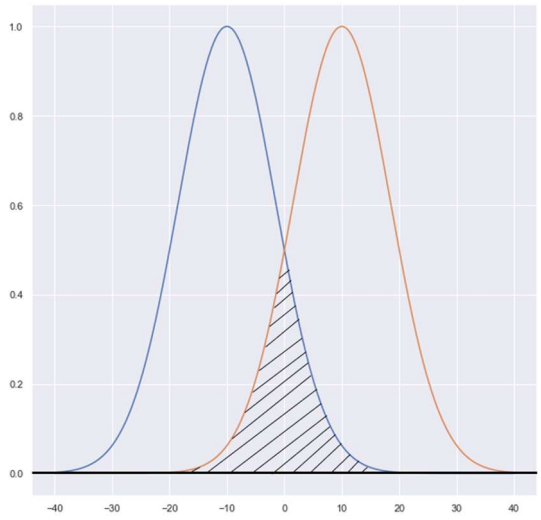
Para se discutir como os conjuntos de lógica *fuzzy* podem ser utilizados de maneira a criar modelos de inteligência artificial, se faz necessário definir algumas funções matemáticas aplicáveis ao conjunto de lógica difusa, como o operador "e".

Na lógica convencional o operador “e” implica a consideração de ambas as premissas de forma igual, isto é, transmite o sentido que a afirmação é verdadeira se e somente se ambas as premissas sejam verdadeiras:

Se fizer sol **e** o céu estiver limpo**, então** irei à praia.

De forma semelhante, na teoria de conjuntos convencional, o operador “e” é substituído pelo operador “∩” (intersecção), que quando operado, tem como resultado os elementos dos conjunto que pertence a ambos os termos operados, conforme a área hachurada na Figura 2.2.

Figura 2.2: Operador intersecção em conjuntos



Fonte: Autor

Vale ressaltar que os conjuntos *fuzzy* possuem as mesmas operações básicas dos conjuntos convencionais, conforme definidas por Zadeh , em 1965 (JANG, 1997 apud TANAKA, 2011). Consequentemente, a fim de realizar a operação “e” no resultado das funções de pertinência, alguns operadores diferentes podem ser considerados. Conforme citado por Passino e Yurkovich (1998), os operadores mais comuns são: a multiplicação de μ_1 e μ_2 ($\mu_1 \cdot \mu_2$) e a função $\min(\mu_1, \mu_2)$, que retorna o menor valor entre μ_1 e μ_2. É importante notar que como ambos μ_1 e μ_2 são limitados ao intervalo [0,1], logo seu produto é obrigatoriamente menor ou igual a μ_1 ou μ_2, mantendo assim o sentido lógico dos conjuntos, afinal “[...] nós não

podemos estar mais certos das conclusões do que estamos das premissas." (PASSINO e YURKOVICH, 1998, p. 42, tradução nossa).

De forma geral, o operador "e" pode ser generalizado na lógica difusa como o operador "$*$" (comumente chamado de norma triangular), que representa ambas as formas de cálculo apresentadas. Logo, $\mu_1(x)*\mu_2(y)$ é utilizado de forma genérica para representar a intersecção de dois conjuntos *fuzzy*.

2.2 Quantificação das variáveis linguísticas

Através das relações estabelecidas para os conjuntos *fuzzy*, se fez possível quantificar as variáveis linguísticas através do uso de regras pautadas na teoria dos conjuntos. Esta é a base do sistema de inferência *fuzzy* (PASSINO e YURKOVICH, 1998 apud TANAKA, 2011).

Para verificar a aplicação das regras com valores linguísticos, é analisado o primeiro exemplo deste capítulo:

Se e somente se o céu estiver limpo, **então** irei à praia.

No conceito de lógica tradicional, considera-se que se a primeira premissa se faz verdadeira, a segunda obrigatoriamente se faz verdadeira. Porém, ao se trazer esta lógica à realidade, se aproximando da forma de raciocínio humano, é possível analisar a seguinte situação: "o céu está parcialmente limpo", que resulta em "talvez eu vá à praia".

Esta forma de raciocínio é chamada de raciocínio difuso, ou raciocínio aproximado (TANAKA, 2011), sendo também empregada quando há mais de uma condição para que a frase seja verdadeira, conforme mencionado na secção 2.1:

Se fizer sol **e** o céu estiver limpo**, então** irei à praia.

Em que é possível inferir uma forma aproximada de raciocínio que propõe que "se fizer muito sol" e "o céu está bastante limpo", "muito provavelmente irei à praia".

Através deste exemplo, pode-se perceber que alterando as variáveis linguisticas, é possível obter resultados aproximados diferentes, como nos dois exemplos a seguir:

Se fizer **muito pouco** sol **e** o céu estiver **bastante** limpo, **provavelmente não** irei à praia.

Se fizer **pouco** sol **e** o céu estiver **parcialmente** limpo, **muito provavelmente não** irei à praia.

Para visualizar essa proposta, é analisado um controlador de velocidade de um carro, em que a variável de entrada é a diferença entre a velocidade do carro e a velocidade desejada, que é chamada de "erro" e o resultado é a intensidade de pressionamento do acelerador. São classificadas as possíveis entradas em cinco intervalos:

Se o erro é **negativo grande**, então pressione **muito mais** o acelerador.

Se o erro é **negativo pequeno**, então pressione **mais** o acelerador.

Se o erro é **nulo**, então **mantenha** o pressionamento do acelerador.

Se o erro é **positivo pequeno**, então pressione **menos** o acelerador.

Se o erro é **positivo grande**, então pressione **muito menos** o acelerador.

A fim de definir uma função de pertinência para cada uma das variáveis linguísticas, apresentadas, é necessário atribuir um valor real para cada variável, que representa o centro da função de pertinência:

-15, se o erro for "**negativo grande**".

-5, se o erro for "**negativo pequeno**".

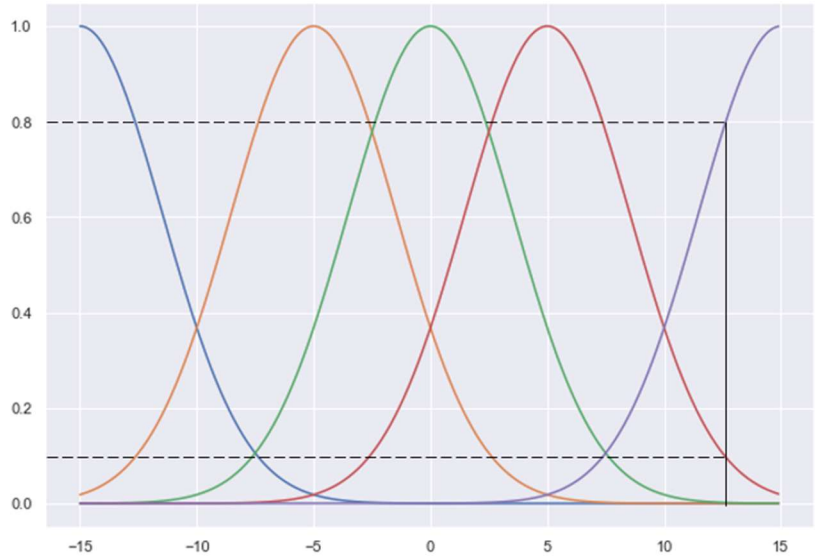
0, se o erro for "**nulo**".

5, se o erro for "**positivo pequeno**".

15, se o erro for "**positivo grande**".

Para cada regra também se atribui um valor real, por exemplo, A = {-2, -1, 0, 1, 2}, respectivamente. A partir das regras definidas, é possível mapear qualquer valor de erro nas variáveis linguísticas utilizando suas funções de pertinência, conforme demonstrado na Figura 2.3, em que as funções de pertinência estão dispostas na ordem da lista acima. Vale ressaltar que as funções de pertinência das extremidades permanecem com seu resultado igual a 1 caso o valor de entrada não esteja entre -15 e 15.

Figura 2.3: Funções de pertinência do controle de velocidade de um automóvel



Fonte: Autor

Utilizando como valor exemplo o erro representado pela linha preta sólida na Figura 2.3, é possível considerar que este erro possui uma pertinência aproximada de 0,8 para a variável linguística “positivo grande” e uma pertinência aproximada de 0,1 para a variável linguística “positivo pequeno”. Isso resulta numa certeza de 0,8 que a regra “se o erro é positivo grande, então pressione muito menos o acelerador” se aplica a esta situação e, semelhantemente, 0,1 de certeza que a regra “Se o erro é positivo pequeno, então pressione menos o acelerador” é aplicável. Realizando a média ponderada dos valores atribuídos às regras, utilizando como peso o grau de certeza de suas premissas, é possível encontrar o valor final esperado para aquela entrada.

Este valor, na situação do controlador de velocidade que foi analisado, é utilizado como entrada para o sistema de operação, como uma voltagem para um motor mecânico ou uma distância a ser percorrida por um atuador. Este processo é chamado *defuzzyficação,* que é abordado na secção 2.3.

Nota-se também através desse exemplo que diferentes valores de entrada podem resultar em diferentes valores totais de pertinência (neste caso, 0,9), um dos fatos que diferencia a abordagem *fuzzy* de uma abordagem probabilística.

2.3 *Defuzzyficação*

Conforme mencionado, após alcançar os resultados das funções de pertinência, se faz necessário realizar o processo de *defuzzyficação,* que objetiva entregar um valor nítido ao sistema operacional. Esse processo pode ser feito de diversas formas, sendo o método TSK (Takagi, Sugeno e Kang) um dos mais o utilizados para isto, que possui a seguinte forma:

Se x é A e y é B, então $z = f(x, y)$

Onde A e B são conjuntos difusos das variáveis de entrada e $z = f(x, y)$ é a função consequente.

Neste trabalho é utilizado apenas o caso representado pela Equação (2.1):

$$z = (p\,x + q\,y + r) \tag{2.1}$$

Sendo p, q e r constantes reais, em que é possível considerar que z se comporta como um interpolador entre os resultados obtidos caso apenas uma regra fosse ativada.

No exemplo da Seção 2.2, foi considerado valores fixos para z, determinados de forma heurística, porém o processo de *defuzzyficação* permanece o mesmo.

Após o cálculo de z_i, sendo i o índice da regra ativada, o resultado nítido do sistema *fuzzy* (Z) é a média ponderada de cada z_i, pelos seu respectivo valor de ativação:

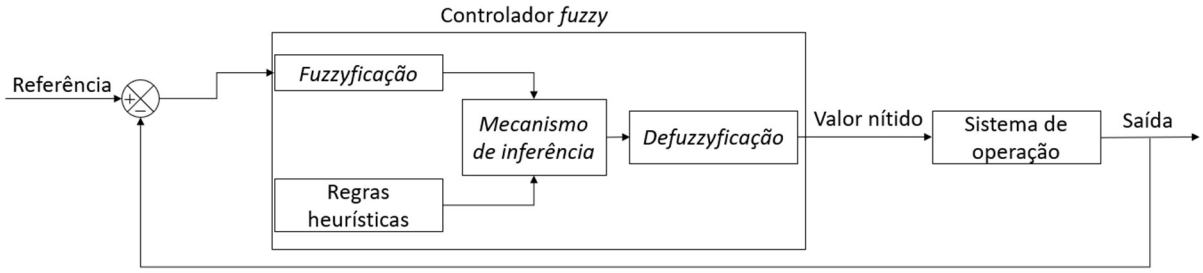
$$Z = \frac{\sum z_i \, \mu_i}{\sum \mu_i} \tag{2.2}$$

Em que μ_i é a pertinência da regra i.

2.4 Sistema *fuzzy*

Considerando todas as etapas demonstradas, é possível sintetizá-las em etapas fundamentais, conforme a Figura 2.4, que as ilustra em um diagrama de blocos para um controlador *fuzzy*.

Figura 2.4: Diagrama de Blocos de um controlador *fuzzy*



Fonte: Autor

É importante notar como através de todo o processo citado diversos parâmetros foram selecionados de forma heurística, como a forma das funções de pertinência, sua quantidade, seus valores de centro, entre outros.

> Basicamente, a escolha de todos os componentes de um controlador *fuzzy* é um tanto *ad hoc*. Quais são as melhores funções de pertinência? Quantas variáveis linguísticas e regras devem existir? Devemos usar a função mínimo

> ou a função produto para representar o ‘e’ na premissa? (PASSINO e YURKOVICH, 1998, p. 34, tradução nossa).

Esse fato ilustra a importância da construção de sistemas adaptativos que busquem otimizar o problema de forma matemática, tirando essa responsabilidade do especialista.

3 SISTEMA DE INFERÊNCIA ADAPTATIVO NEURO-FUZZY

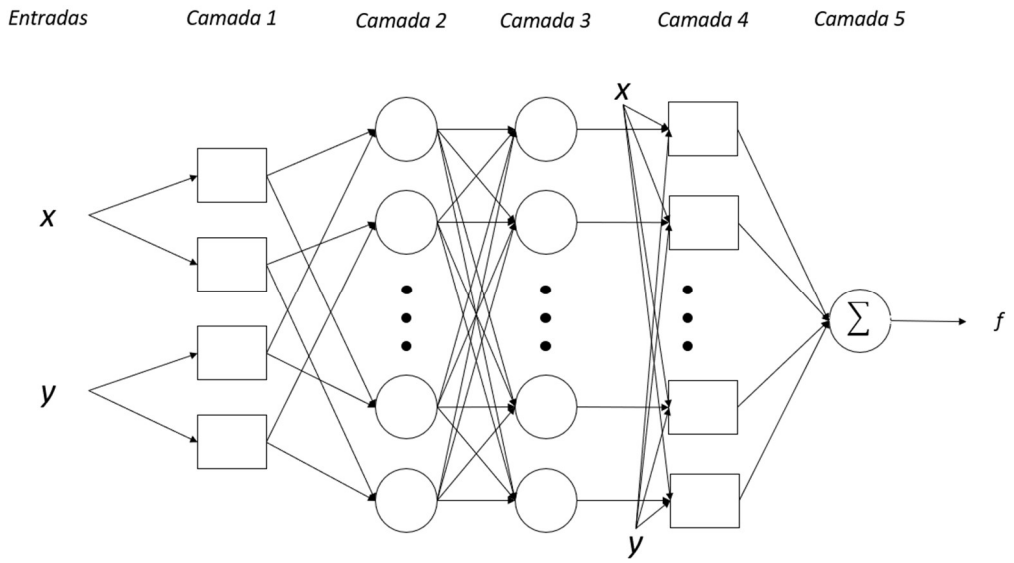
3.1 Arquitetura

Conforme menciona Jang (1993), as redes adaptativas são um superconjunto que engloba todas as redes neurais alimentadas adiante (*feedforward).* Como seu nome implica, as redes adaptativas são redes cuja saída varia de acordo com parâmetros existentes nas suas camadas, que são adaptados com base em uma regra de aprendizagem especificada, a fim de minimizar a diferença entre a saída da rede e o valor esperado.

De forma semelhante, a construção dos sistemas de inferência adaptativos neuro-fuzzy (ANFIS) emprega conceitos utilizados na construção de redes neurais. O ANFIS é constituído por uma rede de múltiplas camadas em que cada nó realiza uma função individual nos sinais de entrada da e nos parâmetros que constituem esse nó. Alguns destes nós, representados como "quadrados" na Figura 3.1, possuem parâmetros que podem ser adaptados, enquanto outros, representados como "círculos", são fixos.

A construção da arquitetura é feita baseado em um sistema de duas variáveis de entrada e uma saída, conforme a Figura 3.1, podendo ser estendida para mais variáveis mediante pequenas alterações.

Figura 3.1: Diagrama de blocos da arquitetura ANFIS



Fonte: Autor

A primeira camada do ANFIS proposta por Jang (1993) realiza o mapeamento das entradas com relação às funções de pertinência escolhidas:

$$Y_{Ai}^{1} = \mu_{Ai}(x_i) \tag{3.1}$$

Sendo x_i a variável de entrada i, μ_{Ai} a função de pertinência da variável linguística A da variável i e Y_{Ai}^{1}, o valor de saída da camada 1 para variável i na função de pertinência da variável linguística A. Por se tratar de um nó variável, os parâmetros que compõem as funções de pertinência devem ser ajustados durante o processo. Os parâmetros dessa camada serão chamados de parâmetros de premissa (JANG, 1993).

Percebe-se ainda que nessa primeira camada há um número de nós equivalente ao somatório das funções de pertinência de cada variável. Considerando-se o caso especial em que o número de variáveis linguísticas é igual para todas as variáveis do sistema, a quantidade de nós é equivalente ao número de variáveis linguísticas multiplicado pelo número de variáveis. Este caso especial é o único utilizado durante o trabalho, a fim de minimizar a complexidade do algoritmo.

A segunda camada realiza a aplicação do operador “e” conforme mencionado na Seção 2.1:

$$Y^2_{AiBj} = W_{AiBj} = \mu_{Ai}(x_i) * \mu_{Bj}(y_j) \tag{3.2}$$

Em que x_i é a variável de entrada i, μ_{Ai} a função de pertinência da variável linguística A da variável i, y_j a variável de entrada j, μ_{Bj} a função de pertinência da variável linguística B da variável j, enquanto W_{AiBj} e Y^2_{AiBj} equivalem à saída da camada 2 para o conjunto da variável i na função de pertinência da variável linguística A com a variável j na função de pertinência da variável linguística B.

Os valores de saída da segunda camada representam a força de ativação das regras linguísticas que cada nó opera (JANG, 1993).

Na segunda camada, utilizando a consideração de que todas as variáveis terão um número igual de funções de pertinência, é obtido um número de nós equivalente ao número de funções elevado ao número de variáveis, o que demonstra que o número de variáveis possui um impacto muito maior quando comparado ao número de variáveis linguísticas na ordem de crescimento (*Big O*) do algoritmo. Essa quantidade de nós permanece igual até a camada 4.

A fim de se aproximar da proposta de Jang (1993), é utilizado a multiplicação para calcular a norma triangular durante a execução deste trabalho.

A terceira camada realiza a normalização das saídas da camada 2 pelo seu somatório:

$$Y^3_{AiBj} = \bar{W}_{AiBj} = \frac{W_{AiBj}}{\sum Y^2} \tag{3.3}$$

Sendo $\sum Y^2$ o somatório de todas as saídas da camada 2 e Y^3_{AiBj} a saída da camada 3 para a combinação da variável i na função de pertinência A com a variável j com a função de pertinência B.

A quarta e a quinta camada, juntas, aplicam o método TSK de *defuzzyficação*, a partir da Função 2.1.

$$Y^4_{AiBj} = \bar{W}_{AiBj}\, f(x,y) = \bar{W}_{AiBj}\,(p_{AiBj}\, x_i + q_{AiBj}\, y_j + r_{AiBj}) \tag{3.4}$$

Sendo x_i e y_j as variáveis iniciais utilizadas para calcular μ_{Ai} e μ_{Bj}, respectivamente, e Y^4_{AiBj} a saída da camada 4 para a combinação da variável *i* na função de pertinência *A* com a variável *j* com a função de pertinência *B*.

Conforme a Figura 3.1, a camada 4 possui seus parâmetros p, q e r variáveis. Estes parâmetros são chamados de parâmetros de consequência, demonstrando a dependência destes parâmetros das funções de pertinência.

De forma semelhante à Seção 2.3, na quinta camada é aplicado a Equação 2.2:

$$Y^5 = \sum \bar{W} f(x,y) = \frac{\sum Wf(x,y)}{\sum Y^2} \tag{3.5}$$

Sendo Y^5 a saída nítida para o sistema de operação.

3.2 Algoritmos de aprendizagem

Neste trabalho é analisado um algoritmo híbrido de aprendizagem, baseado na regra do gradiente reverso e no método dos mínimos quadrados, que visa otimizar o tempo necessário para a função convergir, conforme proposto por Jang (1993).

3.2.1 Gradiente reverso

O método de gradiente reverso, inicialmente proposto por Werbos em 1970 (apud JANG, 1993), propõe que o ajuste de um determinado parâmetro deve ser proporcional à derivada do erro por este parâmetro. Ele é utilizado para calcular a adaptação dos parâmetros de premissa (presentes na Equação 3.1).

Para isto, se faz necessário definir a função de custo (medição de erro) do sistema. Considerando P como o número de dados de treinamento do sistema e p = 1, 2, ..., P, o erro total do sistema equivale a Equação (3.6):

$$E = \sum_{p=1}^{P} E_p \tag{3.6}$$

Em que E_p é o erro do sistema para o dado de treinamento p. De forma semelhante, considerando n o número de nós presente na última camada, sendo m = 1, ..., n, considerando $T_{m,p}$ como o valor desejado da saída m do algoritmo para o dado de treinamento p e considerando $Y_{m,p}^{5}$, a saída m do algoritmo para o dado p, o erro de um dos dados de treinamento pode ser descrito como na Equação (3.7):

$$E_p = \sum_{m=1}^{n} (T_{m,p} - Y_{m,p}^{5})^2 \tag{3.7}$$

Neste trabalho, n na Equação 3.7 é sempre igual a 1 (note que foi definido de forma única na Equação 3.5), ou seja, a camada de saída tem apenas um nó.

Seguindo o raciocínio, a derivada do erro para o dado de treinamento p na última camada equivale a:

$$\frac{\partial E_p}{\partial Y_{m,p}^{5}} = -2(T_{m,p} - Y_{m,p}^{5}) \tag{3.8}$$

A fim de conseguir definir o erro com relação a uma variável em um nó específico, primeiro é preciso definir qual a derivada do erro para um nó interno qualquer. Para isso, considere a camada interna k, com #(k) nós internos, sendo i = 1, ..., #(k), cuja derivada do erro equivale a Equação (3.9):

$$\frac{\partial E_p}{\partial Y_{i,p}^{k}} = \sum_{m=1}^{\#(k+1)} \frac{\partial E_p}{\partial Y_{m,p}^{k+1}} \frac{\partial Y_{m,p}^{k+1}}{\partial Y_{i,p}^{k}} \tag{3.9}$$

Logo, utilizando as Equações 3.8 e 3.9, se faz possível encontrar a derivada do erro de qualquer nó interno. Logo, a derivada do erro para um dado de treinamento p em relação a um parâmetro α pode ser descrita como:

$$\frac{\partial E_p}{\partial \alpha} = \sum_{Y^* \in S} \frac{\partial E_p}{\partial Y^*} \frac{\partial Y^*}{\partial \alpha} \tag{3.10}$$

Sendo α um parâmetro a ser ajustado e S o conjunto de nós cuja saída depende de α. De forma análoga, a derivada do erro total por este parâmetro pode ser descrito como:

$$\frac{\partial E}{\partial \alpha} = \sum_{p=1}^{P} \frac{\partial E_p}{\partial \alpha} \tag{3.11}$$

Assim, a variação sofrida pelo parâmetro α é descrita como:

$$\Delta\alpha = -\eta \frac{\partial E}{\partial \alpha} \tag{3.12}$$

Sendo η definido como:

$$\eta = \frac{k}{\sqrt{\sum_{\alpha} \left(\frac{\partial E}{\partial \alpha}\right)^2}} \tag{3.13}$$

Sendo k a taxa de aprendizagem do algoritmo, definido pelo usuário, podendo ser variada para otimizar o tempo de convergência e evitar que o erro fique preso em mínimos locais (JANG, 1993).

Em sua abordagem, Jang (1993) utiliza o método da propagação resiliente (*resilient-propagation*), que leva em conta a variação do sinal do erro (RIEDMILLER e BRAUN, 1993 apud SILVA *et al*., 2016, p 115). Nessa proposta o valor de *k* deve ser aumentado caso o sinal do erro permaneça igual e deve ser diminuído caso o sinal se inverta, por ser um indicativo de ultrapassado o ponto de mínimo da função erro.

Na abordagem de Jang (1993), é proposto a seguinte heurística, para circundar o fato de sua definição de erro ser feita de forma exclusivamente positiva:

- Se o erro obtido diminuir em quatro gerações seguidas, aumente *k* em 10%.
- Se o erro sofrer duas combinações consecutivas de aumento e diminuição, reduza *k* em 10%.

3.2.2 Método dos mínimos quadrados

Apesar de amplamente utilizado, a velocidade de convergência do algoritmo do gradiente reverso é relativamente baixa, e tende a ficar presa em mínimos locais (JANG, 1993). Devido a isto, um algoritmo de mínimos quadrados sequencial é empregado nos parâmetros de consequência (presentes na Equação 3.4), a fim de encontrar os melhores parâmetros possíveis tendo-se fixado as funções de pertinência.

O método dos mínimos quadrados pode ser interpretado como uma regressão linear para a matriz A, desejando-se definir os melhores valores para a matriz X, com base nos valores de resposta B.

$$AX = B \tag{3.14}$$

A dimensão destas matrizes é respectivamente P x M, M x 1 e P x 1, sendo P o número de dados de treino e M o número de parâmetros lineares (neste caso, parâmetros de consequência), considerando que o sistema proposto tem apenas uma saída por dado de treino.

Considerando que P normalmente é muito maior que M, esse sistema não possui solução exata por ser sobredefinido. Neste caso, o método do mínimo quadrado é idealizado para minimizar a equação $||AX - B||^2$ (JANG, 1993). Por ser um problema conhecido, a equação mais usada é:

$$X^* = (A^T A)^{-1} A^T B \tag{3.15}$$

Sendo X^* a estimativa do mínimo quadrado. Entretanto, quando é considerado a eficiência do código em termos computacionais, realizar a transversa de uma matriz se torna muito custoso. Por isso, visando uma melhora de performance, a forma sequencial é utilizada para o cálculo do mínimo quadrado:

$$S_{i+1} = S_i - \frac{S_i \alpha_{i+1} \alpha_{i+1}^T S_i}{1 + \alpha_{i+1}^T S_i \alpha_{i+1}} \tag{3.16}$$

$$X_{i+1} = X_i + S_{i+1} \alpha_{i+1} (b_{i+1}^T - \alpha_{i+1}^T X_i) \tag{3.17}$$

Onde b_i^T é o elemento da linha i de B, α_i^T é o vetor da linha i de A e $(b_{i+1}^T - \alpha_{i+1}^T X_i)$ pode ser interpretado como o erro do sistema antes do ajuste. Conforme Jang (1993) menciona, as condições iniciais de X_0e S_0 são:

$$X_0 = 0 \tag{3.18}$$

E:

$$S_0 = \gamma I \tag{3.19}$$

Onde I é a matriz identidade de dimensão M x M e γ é um número positivo grande (JANG, 1993) e X_{i+1} é o mínimo quadrado encontrado, após a apresentação de todos os dados de treinamento, naquela geração.

3.3 Utilização do modelo de aprendizagem

No modelo de aprendizagem proposto por Jang (1993), o método dos mínimos quadrados é realizado uma vez por geração, na etapa de propagação adiante, visando determinar quais são os melhores parâmetros para o método TSK de *defuzzyficação* dado as funções de pertinência, que permanecem fixas nessa etapa. De forma semelhante, os parâmetros das funções de pertinência (parâmetros de premissa) são adaptados na etapa de propagação reversa (*backpropagation*) através do método do gradiente reverso, enquanto os parâmetros de consequência são mantidos fixos.

4 DESENVOLVIMENTO DA ARQUITETURA UTILIZADA

Nesta seção é abordado como os parâmetros do sistema serão variados para as análises presentes no capítulo 5. Para utilização deste sistema, faz-se necessário definir o espaço de operação do sistema, que por sua vez define os centros das funções de pertinência, espaçando-as de forma igual. Após esta definição, é necessário escolher os demais parâmetros do sistema, conforme comentado na seção 4.1, assim como o número de dados de treinamento e validação e o número máximo de gerações. Este passo a passo deve ser repetido para cada sistema que se deseje utilizar o ANFIS.

4.1 Determinação das variáveis do sistema

Utilizando como base o trabalho proposto por Jang (1993), neste capítulo é desenvolvido a arquitetura utilizada para obter os resultados e considerações finais. Isso se deve ao fato de o ANFIS ainda possuir diversas decisões que estão a cargo do desenvolvedor de sua arquitetura. Entre elas é possível citar:

- Qual a função de pertinência a ser utilizada;
- O número de variáveis linguísticas;
- O intervalo de operação;
- Os parâmetros iniciais da função de pertinência;
- O operador utilizado para a norma triangular;
- O valor inicial de γ, na Equação 3.19;
- O valor da taxa de aprendizagem;
- A utilização ou não do algoritmo de propagação resiliente;
- A proporção em que se deve aumentar e diminuir a taxa de aprendizagem com o algoritmo de propagação resiliente;

- A quantidade de dados de treinamento;
- O número de gerações de treinamento.

A fim de definir todos os parâmetros, se faz necessário realizar algumas decisões de projeto. Conforme comentado no capítulo 1, o problema proposto para análise é um sistema seno cardinal de duas variáveis, que pode ser definido pela seguinte equação:

$$Z = (\frac{\sin(x)}{x})(\frac{\sin(y)}{y}) \tag{4.1}$$

Sendo o seno de *x* e *y* calculados em radianos e *x* e *y* pertencentes ao intervalo [-10,10], que é o espaço de operação do sistema proposto (JANG, 1993).

A função de pertinência utilizada por Jang (1993) é uma função com forma de sino definida por:

$$\mu_A(x) = \frac{1}{1+[(\frac{x-c}{a})^2]^b} \tag{4.2}$$

Porém, neste trabalho é utilizado a função Gaussiana definida por:

$$\mu_A(x) = b \exp(-(\frac{x-c}{a})^2) \tag{4.3}$$

Devido a sua derivada ser sempre uma função com forma de sino. Como as funções de pertinência devem ser limitadas entre [0,1] conforme mencionado no capítulo 2, neste trabalho *b* é constante e igual a 1.

A variável *a*, que representa a meia largura da função, sendo definida de uma forma um tanto *ad hoc,* buscando apenas garantir que o valor de pertinência inicial para qualquer valor dentro do intervalo de operação escolhido seja pelo menos 0,5 (JANG, 1993). Neste trabalho, o valor de *a* é fixado em 12, valor mínimo para se alcançar o requisito citado para todas as variações analisadas, sendo que sua influência no resultado não é analisada.

A variável *c,* que representa o centro das funções de pertinência, que é definida de forma a dividir o espaço de operação de forma simétrica, com base no número de variáveis linguísticas.

O número de variáveis linguísticas é um dos parâmetros analisados neste trabalho. Tendo sido inicialmente proposto por Jang (1993) apenas um sistema de quatro variáveis linguísticas para a função seno cardinal, afirmando que os sistemas de duas e três variáveis não eram robustos o suficiente para descrever um sistema altamente não linear. Neste trabalho são analisados sistemas de 2 a 7 variáveis linguísticas.

Conforme utilizado por Jang (1993), a operação que representa o operador linguístico "e" neste trabalho é apenas a multiplicação.

O valor inicial de γ, presente na Equação 3.19 não é mencionado por Jang (1993), havendo apenas o comentário da necessidade de ser um número positivo grande. Neste trabalho não é analisado o impacto da variação do valor de γ nos resultados de erro, sendo utilizado:

$$\gamma = 10^6 \tag{4.4}$$

Em Jang (1993), o valor da taxa de aprendizagem é variado de 0,01 a 0,1, obtendo 10 diferentes taxas de aprendizagem sob as quais é generalizado um gráfico da raiz do erro quadrático médio por geração, para realizar a comparação com um algoritmo RNA com algoritmo de aprendizagem *quickpropagation* (propagação rápida), sendo considerado em todas as iterações os mesmos dados de treinamento, sem apresentação de dados de checagem. Esta análise de performance não é realizada neste trabalho, sendo o valor inicial da taxa de aprendizagem fixado em 0,1 em todas as iterações.

A utilização do método do algoritmo de aprendizagem resiliente, entretanto, é analisado neste trabalho, sendo realizadas análises do impacto de seus parâmetros constituintes no resultado obtido.

Conforme comentado, na análise proposta por Jang (1993) todos os procedimentos realizados utilizaram com o mesmo conjunto de 121 dados de treinamento. Nesta análise, são utilizados conjuntos de dados iniciais aleatórios de igual tamanho para realizar o treino do algoritmo, verificando sua capacidade de generalização por meio de um conjunto de checagem aleatório.

4.2 Equação do termo de ajuste do algoritmo de gradiente reverso

A fim de encontrar o fator de ajuste dos parâmetros dos parâmetros de premissa, descritos na Equação 3.1, se faz necessário calcular os termos presentes na Equação 3.10.

Inicialmente, a derivada da função de pertinência por suas constantes *a* e *c* são respectivamente:

$$\frac{\partial}{\partial a}\exp\left(-\left(\frac{x-c}{a}\right)^2\right) = \frac{2(x-c)^2\left(\exp\left(-\left(\frac{x-c}{a}\right)^2\right)\right)}{a^3} \tag{4.5}$$

$$\frac{\partial}{\partial c}\exp\left(-\left(\frac{x-c}{a}\right)^2\right) = \frac{2(x-c)\left(\exp\left(-\left(\frac{x-c}{a}\right)^2\right)\right)}{a^2} \tag{4.6}$$

Sendo ambas referidas a partir desse ponto como $\frac{\partial Y_{ai}^1}{\partial a}$, sendo Y_{ai}^1 a saída do nó da primeira camada depende da variável de entrada *i* cuja função possui o parâmetro a, conforme notação utilizada nas Equações 3.1 e 3.10, a fim de generalizar os resultados obtidos para todo ANFIS que segue o modelo proposto por Jang (1993).

A fim de encontrar a derivada do erro pela última camada, partir da Equação 3.8, considerando $n = 1$, é obtido:

$$\frac{\partial E_p}{\partial Y_p^5} = -2(T_p - Y_p^5) \tag{4.7}$$

Da Equação 3.9 e 3.10, como todas as camadas são dependentes das funções de pertinência, é obtido:

$$\frac{\partial E_p}{\partial \alpha} = \frac{\partial E_p}{\partial Y^5}\frac{\partial Y^5}{\partial Y^4}\frac{\partial Y^4}{\partial Y^3}\frac{\partial Y^3}{\partial Y^2}\frac{\partial Y^2}{\partial Y_{ai}^1}\frac{\partial Y_{ai}^1}{\partial \alpha} \tag{4.8}$$

Realizando todas as derivadas no modelo do ANFIS proposto por Jang (1993), para qualquer $Y_{ai}^1 \neq 0$ e $\overline{W} \neq 0$, é obtido:

$$\frac{\partial E_p}{\partial \alpha} = \left(\frac{\partial Y_{ai}^1}{\partial \alpha}\right)\left(\frac{1}{\bar{W} Y_{ai}^1}\right)[\sum_{ai}(p_{ai}x_p + q_{ai}y_p + \cdots + r_{ai} - Y_p^5)W_{ai}] \quad (4.8)$$

Sendo *ai* o representativo de todos os termos que dependem de Y_{ai}^1 , W_{ai} todas as saídas de Y^2 que dependam de Y_{ai}^1 , *p, q* e *r* os parâmetros de consequência da Equação 3.4, x_p e y_p as variáveis de entrada sistema e $\bar{W}$ o somatório de todas as saídas de Y^2.

Entretanto a Equação 4.8 tem pouca eficiência computacional, pois não há nada que impeça que Y_{ai}^1 seja igual a zero. Logo, W_{ai} deve ser substituído pela combinação linear de todas as regras de todas as variáveis de entrada, excetuando a variável analisada *i*, equivalente a $\frac{W_{ai}}{Y_{ai}^1}$, para $Y_{ai}^1 \neq 0$. As reticências visam demonstrar que essa equação é geral para qualquer número de variáveis de entrada, apesar de apenas duas serem utilizadas neste trabalho.

5 RESULTADOS E DISCUSSÕES

Nesta seção são apresentados os resultados obtidos a partir do treinamento do ANFIS com base nos critérios mencionados na Seção 4.1. Esta seção é dividida em três partes: análise da variação do número de variáveis linguísticas; análise do algoritmo de propagação resiliente e seus parâmetros; análise do melhor modelo obtido.

Conforme citado, não é estipulado condição de parada com base no erro mínimo, nem é utilizado critério de parada antecipada, sendo o único critério de parada o número de gerações da rede, fixada em 250 gerações, a fim de manter certa proximidade com o trabalho de Jang (1993) e por ser observado que não há melhora significativa na utilização de mais gerações. Apesar disso, é utilizado o conceito de validação cruzada, escolhendo-se a geração com menor valor de raiz do erro quadrático médio (*root mean squared error,* RMSE) no conjunto de validação. O RMSE do conjunto de validação é o único utilizado para criação dos gráficos e tabelas.

Conforme mencionado, neste trabalho são utilizados 121 dados de treinamento (45% dos dados totais), junto a um conjunto de validação de 151 dados aleatórios (55% dos dados totais), exceto se mencionado de forma diferente na apresentação dos dados.

As imagens tridimensionais geradas utilizam um conjunto aleatório de 2000 dados de checagem, a fim de se obter uma imagem com alto número de polígonos.

5.1 Análise da variação do número de variáveis linguísticas

Nesta seção é realizado o treinamento de 6 sistemas com diferentes quantidades de variáveis linguísticas, sendo o menor com duas variáveis e o maior com 7.

Para isto, o algoritmo de propagação resiliente é utilizado conforme a seguinte heurística:

- Se o RMSE obtido diminuir em quatro gerações seguidas, aumente *k* em 1%;
- Se o RMSE sofrer um aumento e uma diminuição, diminua *k* em 10%.

A escolha dos parâmetros do algoritmo de propagação resiliente é descrito na Seção 5.2. Seu ajuste é realizado antes da propagação reversa de cada geração.

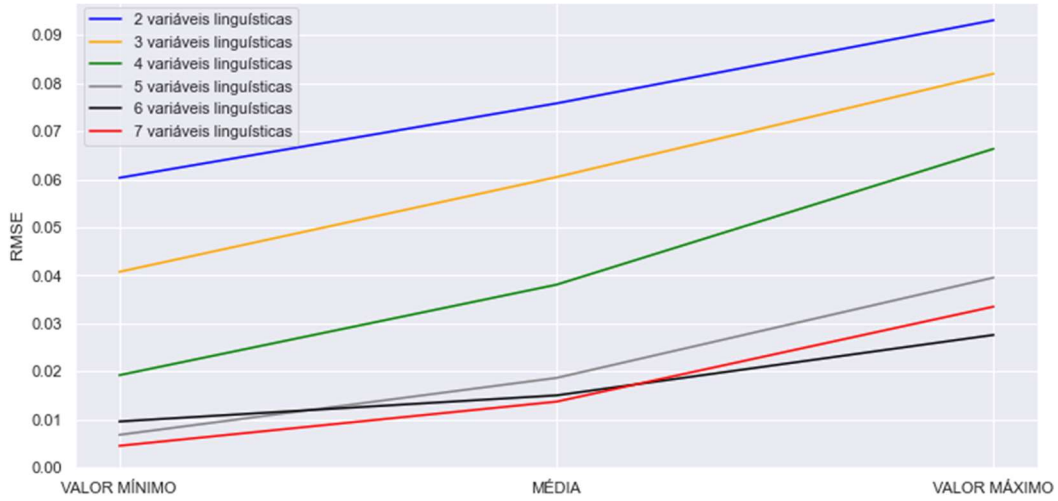
O número de variáveis linguísticas, junto com o RMSE mínimo, médio e máximo do conjunto de validação são mostrados na Tabela 5.1 e na Figura 5.1.

Tabela 5.1: Valores de RMSE mínimo, médio e máximo por número de variáveis linguísticas

Nº de variáveis linguísticas	RMSE mínimo	RMSE médio	RMSE máximo
2	0,06029	0,07579	0,09308
3	0,04074	0,06045	0,08196
4	0,01925	0,03808	0,06634
5	0,00685	0,01869	0,03955
6	0,00963	0,01506	0,02763
7	0,00457	0,01375	0,03352

Fonte: Autor

Figura 5.1: Gráfico de RMSE mínimo, médio e máximo por número de variáveis linguísticas

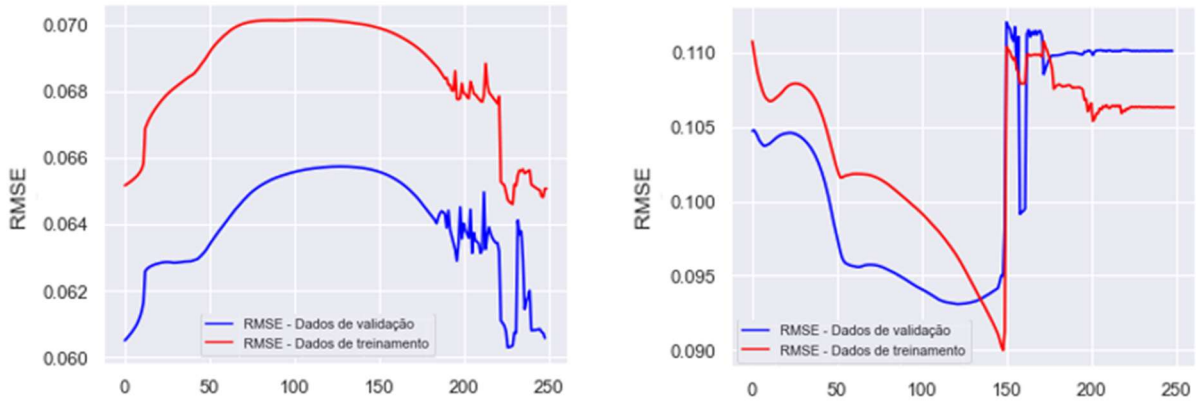


Fonte: Autor

Chama a atenção a forma de disposição das curvas na Figura 5.1, deixando claro a influência da quantidade de variáveis linguísticas a fim de modelar o problema. Entretanto, percebe-se a partir de 5 variáveis linguísticas, o RMSE teve uma queda reduzida, demonstrando que há um limite para essa conclusão.

Jang (1993) afirma que para a função modelada, os sistemas de duas e três funções de pertinência não possuem complexidade suficiente para modelar o problema. Entretanto, conforme pode ser verificado nas Figuras 5.2 e 5.3, que demonstra a melhor e a pior iteração para cada um dos dois sistemas citados, essa conclusão pode ser reavaliada. Em ambas as figuras, a linha vermelha representa o RMSE durante o treinamento e a linha azul o RMSE durante a checagem. As Figuras 5.2 (a) e 5.3 (a) demonstra as curvas produzidas pelos conjuntos de dados que obtiveram menor RMSE no conjunto de validação, enquanto as Figuras 5.2 (b) e 5.3 (b) demonstram as curvas que obtiveram maior RMSE.

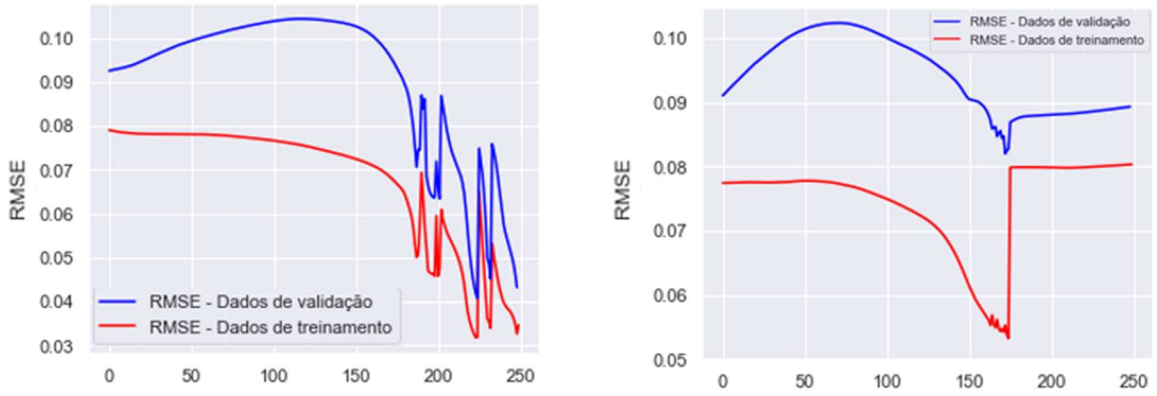
Figura 5.2: Curva de RMSE por geração para 2 variáveis linguísticas



(a) Condição de menor RMSE (b) Condição de maior RMSE

Fonte: Autor

Figura 5.3: Curva de RMSE por geração para 3 variáveis linguísticas



(a) Condição de menor RMSE (b) Condição de maior RMSE

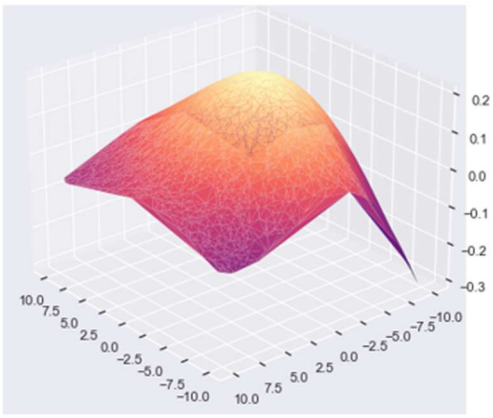
Fonte: Autor

Nota-se para o sistema com apenas duas variáveis que na consideração do RMSE mínimo, o RMSE de checagem é exclusivamente menor que o RMSE de treino e que apesar de possuir um ponto de mínimo diferente do ponto inicial, a maior parte das gerações apenas causaram um aumento do RMSE. Semelhante é observado na condição de RMSE máximo, entretanto, com a convergência do RMSE a um valor de mínimo com o passar das gerações.

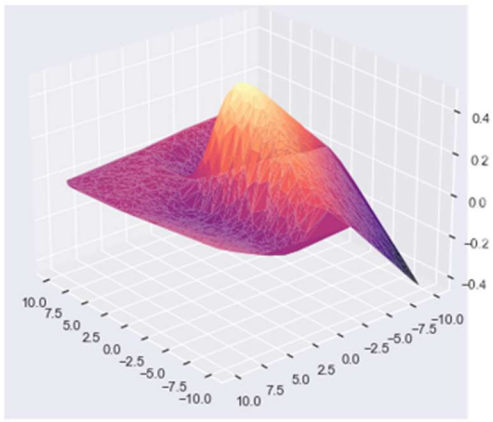
Já no sistema com três variáveis linguísticas, é observado um comportamento mais comum, em que o sistema tende a convergir a um RMSE mínimo, tanto no treino quanto na checagem.

Outro resultado que pode ser analisado para distinguir o avanço da rede com as gerações é comparar a curva no espaço estimado pela rede na geração 0, em que apenas os mínimos quadrados são calculados, podendo ser comparado a um sistema *fuzzy* comum, com a sua melhor geração e com a curva real do sistema. As Figuras 5.4 e 5.5 demonstram esse avanço para a situação de RMSE máximo.

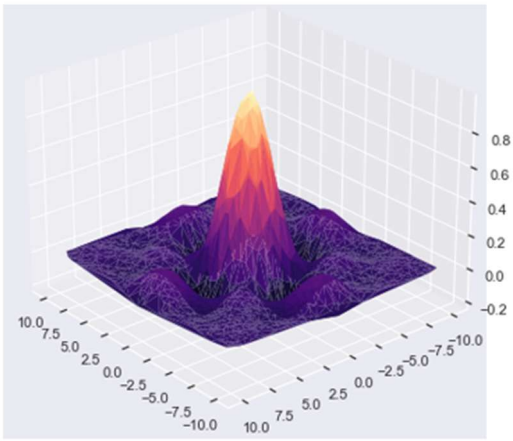
Figura 5.4: Curva seno cardinal no espaço para 2 variáveis linguísticas



(a) Aproximação da curva: geração 0



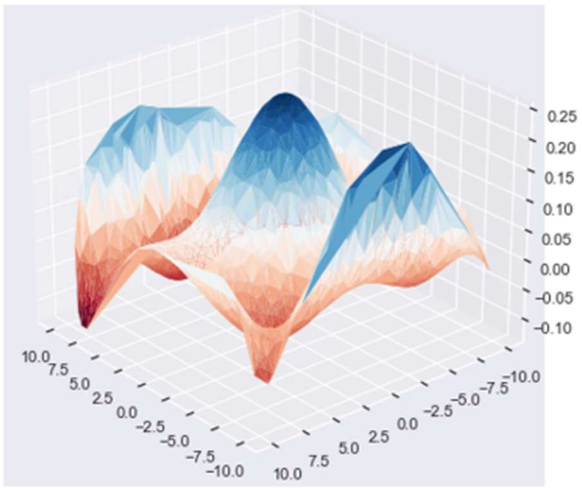
(b) Aproximação da curva: melhor geração



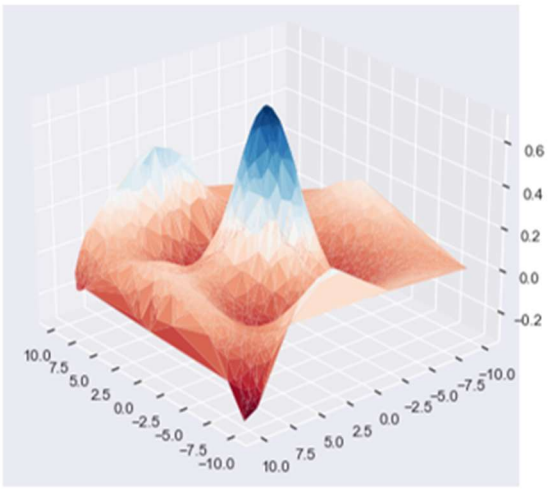
(c) Curva seno cardinal

Fonte: Autor

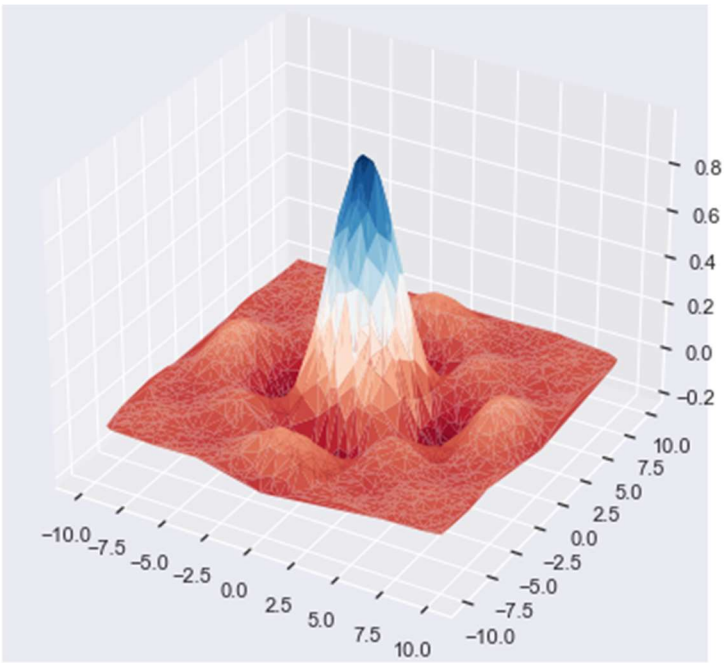
Figura 5.5: Curva seno cardinal no espaço para 3 variáveis linguísticas



(a) Aproximação da curva: geração 0



(b) Aproximação da curva: melhor geração



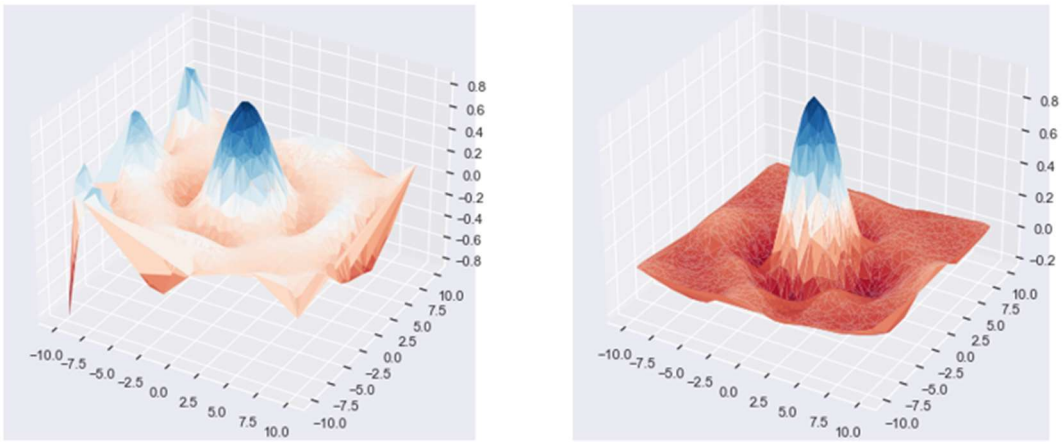
(c) Curva seno cardinal

Fonte: Autor

Através da Figura 5.5, é possível notar o avanço do sistema com três variáveis linguísticas, denunciando também a precariedade do modelo gerado pelo sistema de duas variáveis.

De forma semelhante, as Figura 5.6 e 5.7 demonstram a melhor e a pior aproximação gerada pelos sistemas de 4 e 7 para o conjunto de dados apresentados.

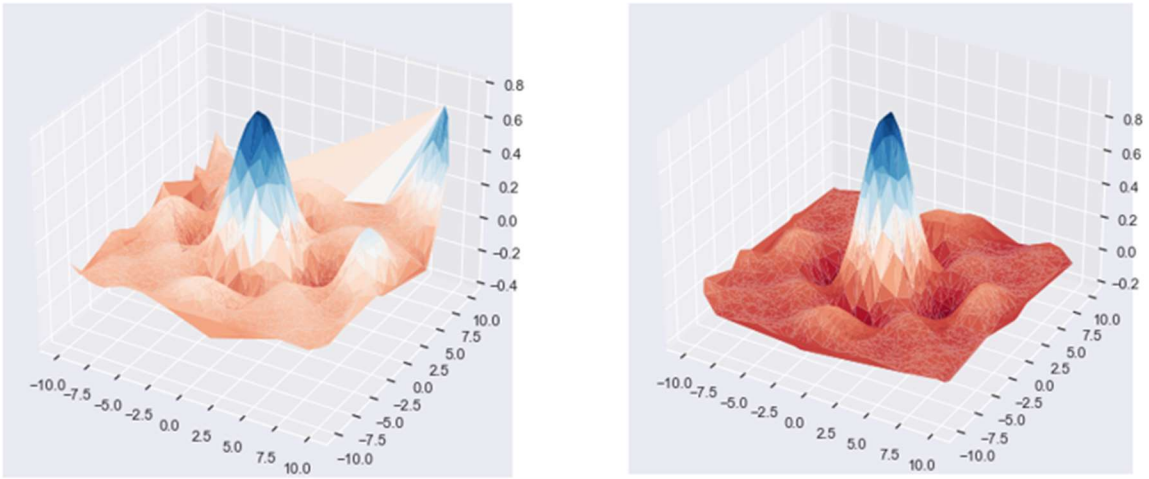
Figura 5.6: Curva seno cardinal no espaço para 4 variáveis linguísticas



(a) Aproximação da curva: maior RMSE (b) Aproximação da curva: menor RMSE

Fonte: Autor

Figura 5.7: Curva seno cardinal no espaço para 7 variáveis linguísticas



(a) Aproximação da curva: maior RMSE (b) Aproximação da curva: menor RMSE

Fonte: Autor

Através destas figuras, é possível concluir que ambos os modelos conseguiram aproximar a curva desejada, confirmando os dados da Tabela 5.1.

5.2 Análise do algoritmo de propagação resiliente

Conforme mencionado na Seção 3.2.1, devido à definição da função de custo descrita por Jang (1993) ser estritamente positiva, as regras para construção do algoritmo de propagação resiliente são:

- Se o RMSE obtido diminuir em quatro gerações seguidas, aumente *k* em 10%;
- Se o RMSE sofrer duas combinações consecutivas de aumento e diminuição, diminua *k* em 10%.

Entretanto, Jang (1993) menciona que estes parâmetros de construção foram escolhidos de forma relativamente arbitrária, apesar de entregarem resultados satisfatórios.

Podem ser destacados os seguintes parâmetros:

- Número de iterações de redução seguidas para incremento de *k*;
- Valor de incremento de *k*;
- Número de combinações consecutivas de redução e aumento do RMSE para reduzir o valor de *k*;
- Valor de decremento de *k*.

Neste trabalho, são analisados os impactos dos dois primeiros parâmetros citados. Desta forma, é fixado uma combinação de uma redução e um incremento de RMSE para se reduzir o valor de *k* em 10% em todas as análises.

5.2.1 Análise do número de iterações do algoritmo de propagação resiliente

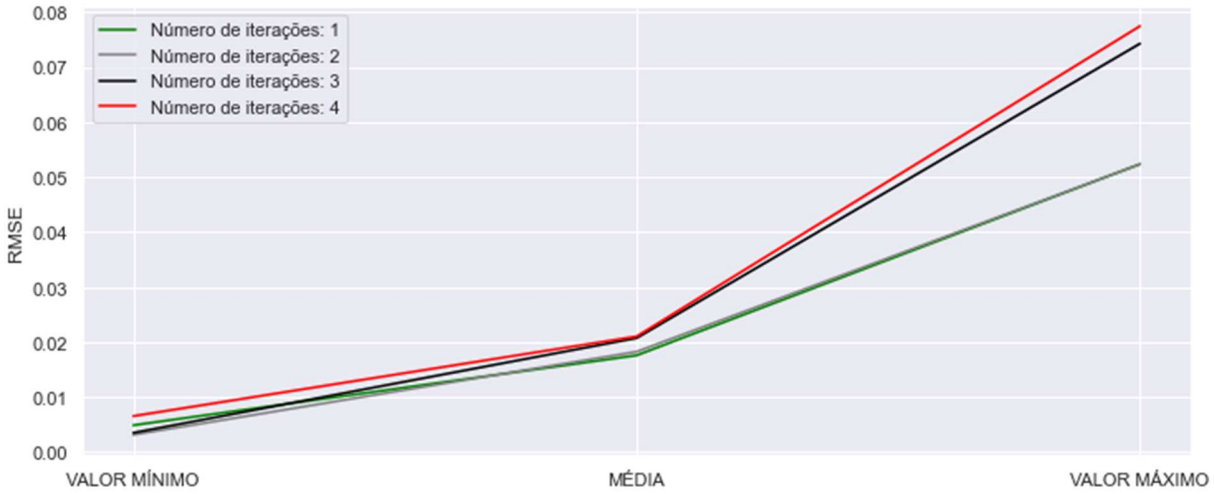
Nesta seção é realizada a comparação de vinte conjuntos de dados aleatórios aplicados aos sistemas com 4, 6 e 7 variáveis, variando-se entre um e quatro o número de iterações necessárias para incrementar o valor de *k* em cada geração. Isto fornecendo as informações para construção da Tabela 5.2 e da Figura 5.8. O valor de incremento é fixado em 1% do valor de *k* para esta análise.

Tabela 5.2: Valores de RMSE mínimo, médio e máximo por número de iterações necessárias para incremento do valor de *k*

Número de iterações	RMSE mínimo	RMSE médio	RMSE máximo
1	0,00488	0,01758	0,05235
2	0,00314	0,01824	0,05232
3	0,00350	0,02073	0,07424
4	0,00654	0,02103	0,07742

Fonte: Autor

Figura 5.8: Gráfico de RMSE mínimo, médio e máximo por número de iterações necessárias para incremento do valor de *k*



Fonte: Autor

Nota-se através da Figura 5.7 que existe uma correlação entre a diminuição do RMSE e o número de iterações necessárias para o incremento do valor de *k*, utilizando-se o valor de incremento igual a 1% do valor de *k*. A porcentagem de diminuição de RMSE, com base no valor de RMSE médio é de 16%, alcançando 52% no mínimo e 32% no máximo.

5.2.2 Análise do incremento por iteração do algoritmo de propagação resiliente

A fim de analisar a contribuição do valor de incremento de *k* no RMSE, na Tabela 5.3 e Figura 5.9 é demonstrado os valores obtidos da análise considerando vinte conjuntos de dados aleatórios, aplicado apenas para um número de variáveis linguísticas igual a 7 e considerando duas iterações de redução de RMSE necessárias para se alterar o valor de *k,* visto que este número de iterações possui o menor valor de mínimo na análise anterior e valores de média e de máximo comparáveis à condição de uma iteração.

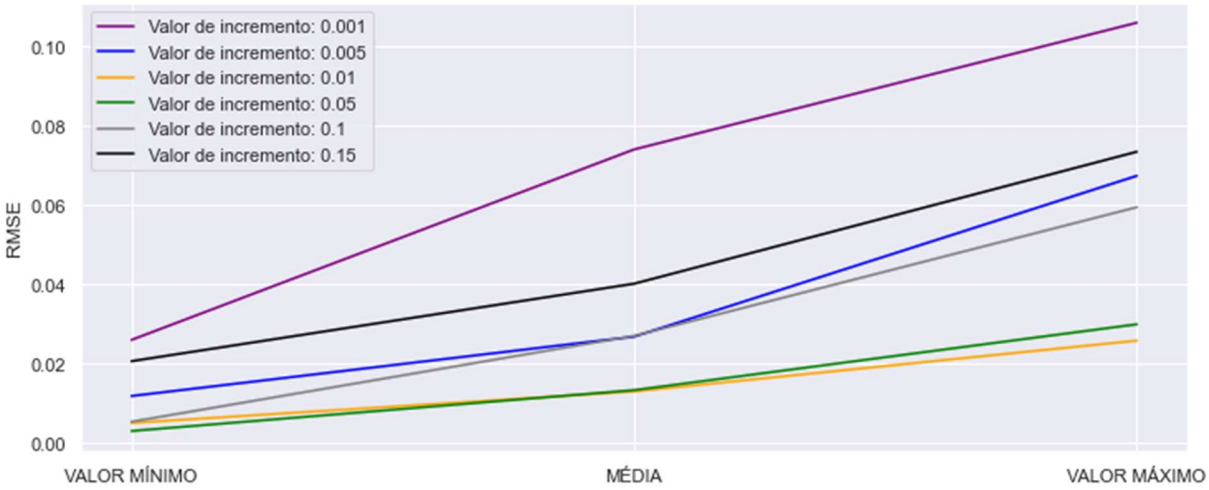
Os valores de incremento são representados como porcentagens do valor de *k.*

Tabela 5.3: Valores de RMSE mínimo, médio e máximo para diferentes incrementos

Valor de incremento	RMSE mínimo	RMSE médio	RMSE máximo
0,1%	0,02607	0,07409	0,10602
0,5%	0,01193	0,02690	0,06743
1,0%	0,00514	0,01308	0,02589
5,0%	0,00312	0,01342	0,03001
10,0%	0,00545	0,02710	0,05952
15,0%	0,02071	0,04024	0,07350

Fonte: Autor

Figura 5.9: Gráfico de RMSE mínimo, médio e máximo para diferentes porcentagens de incrementos

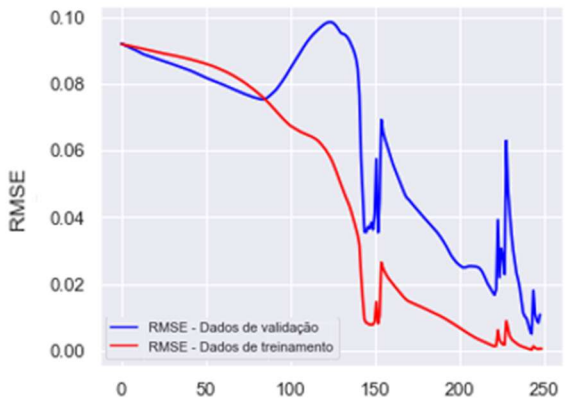


Fonte: Autor

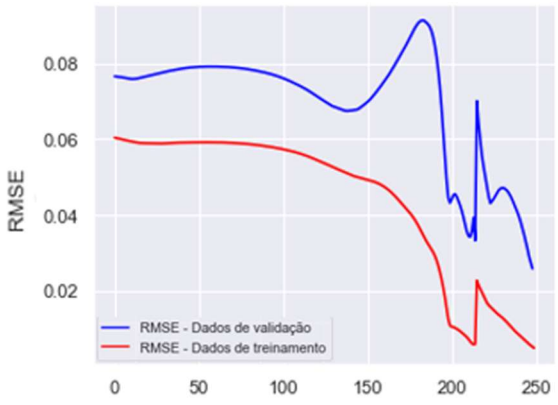
A Figura 5.9 demonstra claramente que é alcançado um RMSE médio mínimo quando o valor de incremento aproxima de 1 e 5%, demonstrando que existe uma correlação entre o RMSE e o valor de incremento do algoritmo de propagação resiliente.

Comparando-se ainda os resultados das duas configurações com melhor performance, na Figura 5.10, é possível notar como a aproximação gerada pelo incremento de 1% ocorreu de forma mais suave que o incremento de 5%.

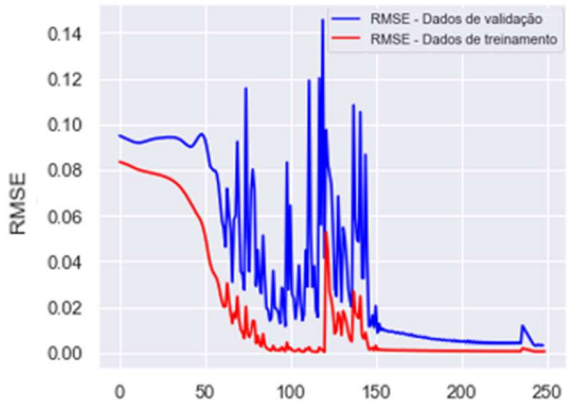
Figura 5.10: Curva de RMSE por geração das configurações com 1 e 5% de incremento



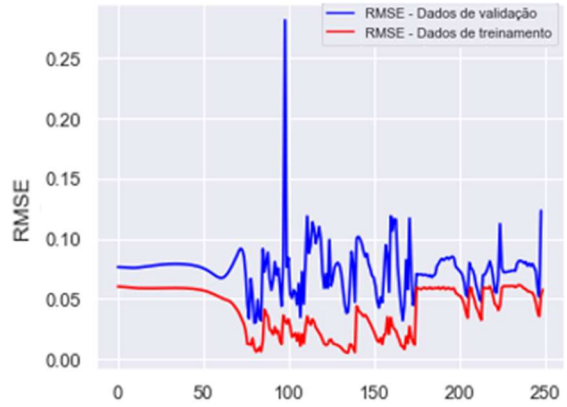
(a) Condição de menor RMSE para 1%



(b) Condição de maior RMSE para 1%



(d) Condição de menor RMSE para 5%



(b) Condição de maior RMSE para 5%

Fonte: Autor

5.3 Análise dos melhores resultados

Considerando todos os parâmetros analisados, os valores de mínimo, máximo e média de RMSE e a curva no espaço do melhor modelo alcançado com os parâmetros ótimos obtidos, podem ser analisados na Tabela 5.4 e na Figuras 5.12. Esse modelo considera 7 variáveis

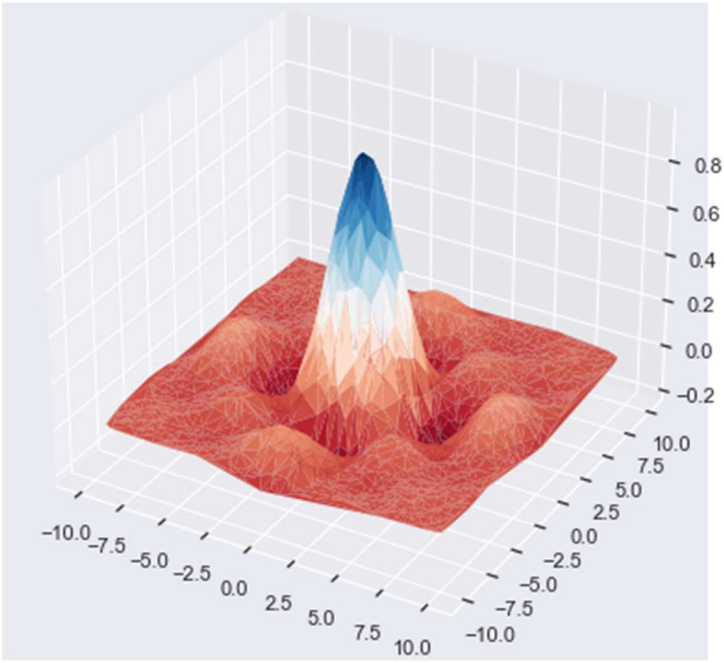
linguísticas, com valor de incremento do algoritmo de propagação resiliente igual a 1% para duas iterações de redução em quarenta conjuntos de dados iniciais diferentes (vinte da Seção 5.2.1 e vinte da Seção 5.2.2). A Figura 13 demonstra a primeira e a melhor geração do mesmo sistema, alterando-se apenas o valor de incremento para 5%. Os demais parâmetros fixados anteriormente são mantidos constantes.

Tabela 5.4: Valores de RMSE mínimo, médio e máximo para o modelo alcançado

RMSE mínimo	RMSE médio	RMSE máximo
0,00314	0,01264	0,03593

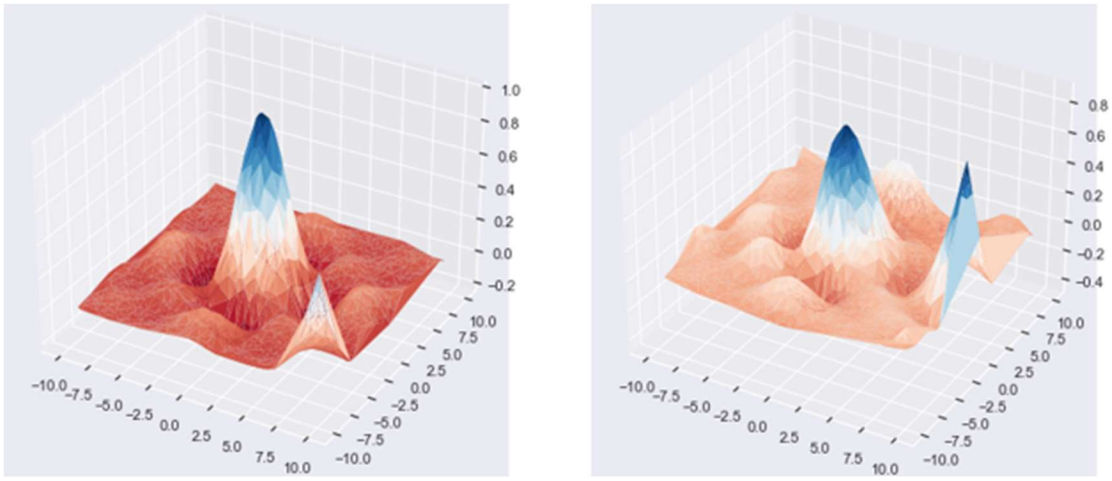
Fonte: Autor

Figura 5.11: Curva seno cardinal



Fonte: Autor

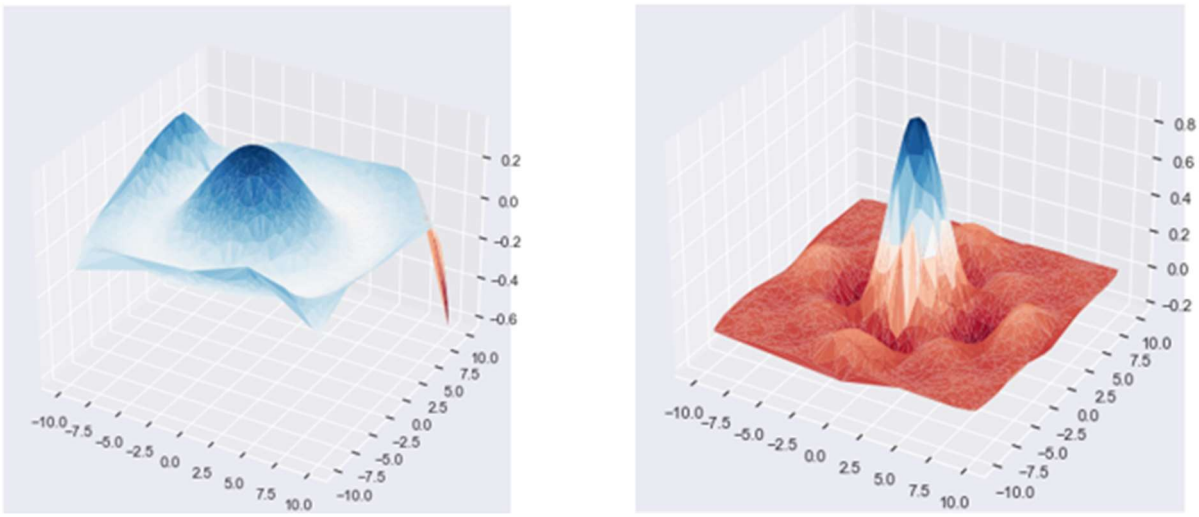
Figura 5.12: Curva seno cardinal para o melhor modelo com 1% de incremento



(a) Aproximação da curva: menor RMSE (b) Aproximação da curva: maior RMSE

Fonte: Autor

Figura 5.13: Curva seno cardinal para o modelo com 5% de incremento



(a) Aproximação da curva: geração 0 (b) Aproximação da curva: melhor geração

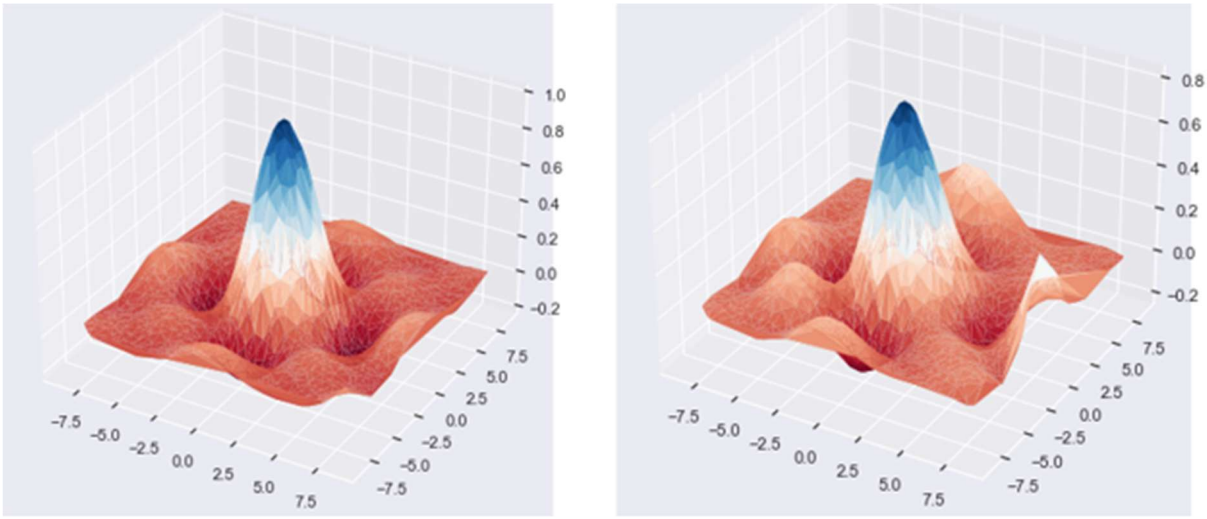
Fonte: Autor

É notório que ambos os sistemas foram capaz de aproximar a função desejada. Também é possível concluir que o sistema em sua condição inicial, que pode ser comparada

a um sistema fuzzy não adaptativo com poucas informações iniciais, teve uma melhora significativa em sua performance, comprovando a qualidade e o objetivo do ANFIS.

Também é notório nas figuras de aproximação da curva seno cardinal no espaço demonstradas neste capítulo que as regiões com maiores erros acontecem próximos aos limites do intervalo de operação do modelo. Desta forma, é possível concluir que diminuir o intervalo de operação em relação ao intervalo de treino pode melhorar a performance do modelo. Jang (1993) utiliza esse método em um sistema não linear de três variáveis, tornando o intervalo de operação menor que o utilizado para treino. A fim de visualizar a diferença entre os modelos, a Figura 5.14 demonstra a aproximação da curva seno cardinal para valores randômicos entre -8,5 e 8,5.

Figura 5.14: Curva seno cardinal para o modelo alcançado no intervalo de [-8,5 , 8,5]



(a) Aproximação da curva: menor RMSE (b) Aproximação da curva: maior RMSE

Fonte: Autor

Com essa modificação, é obtido um RMSE mínimo de 0,00245 e um máximo de 0,02930, para um intervalo de teste de dois mil dados entre -8,5 e 8,5.

6 CONSIDERAÇÕES FINAIS E PERSPECTIVAS FUTURAS

Neste trabalho foi realizada uma análise do sistema de inferência adaptativo *neuro-fuzzy* (ANFIS) proposto por Jang (1993), construído sob a base dos sistemas de lógica *fuzzy*, conforme discutido nos capítulos 2 e 3, que utiliza uma lógica de aprendizagem de máquina baseada nos algoritmos de gradiente reverso e mínimos quadrados.

Através da análise exaustiva de alguns dos seus parâmetros constituintes, são realizados otimizações nos conceitos originais propostos por Jang (1993) e discutido alguns dos resultados concluídos por ele, com base na aplicação em um sistema seno cardinal de duas variáveis.

Entre os pontos discutidos estão a capacidade de generalização de sistemas com apenas três variáveis independentes, possibilidade que foi descartada por Jang (1993) para o sistema seno cardinal em questão, e a confirmação da precariedade do modelo gerado pelo sistema com apena duas variáveis.

Analisando as curvas aproximadas pelos ANFIS da função seno cardinal de duas variáveis, é possível perceber como o sistema conseguiu, na maioria dos casos, capturar as características específicas do sistema analisado, melhorando significativamente sua performance quando comparado a um sistema *fuzzy* não adaptativo com as mesmas condições iniciais. Em relação ao trabalho de Jang (1993), temos a expansão dos resultados através da utilização de conjuntos de validação, aplicação a mais de um conjunto de treinamento, considerações sobre a capacidade de generalização dos sistemas e a análise do impacto do número de variáveis linguísticas no resultado obtido.

Foram analisados também os efeitos de das variáveis do algoritmo de propagação resiliente, sendo concluído que estes parâmetros podem ter um resultado significativo na análise, e definindo o melhor modelo para o sistema seno cardinal de duas variáveis.

Para os trabalhos futuros, é proposto a aplicação destes conceitos nas demais variáveis não analisadas, como na condição inicial de α e de γ, presentes nas Equações 4.2 e 3.19,

respectivamente. Também é sugerido a aplicação dos conceitos em sistemas mais complexos, como sistemas de três variáveis independentes e sistemas reais de controle, a fim de verificar se os resultados obtidos podem ser generalizados.

REFERÊNCIAS

JANG, J. S. R. ANFIS: adaptive-network-based fuzzy inference system. In: **IEEE Transactions on Systems, Man, and Cybernetics**, v. 23, n. 3, p. 665-685, May/June 1993. Disponível em: https://ieeexplore-ieee-org.ez10.periodicos.capes.gov.br/stamp/stamp.jsp?tp=&arnumber=256541. Acesso em: 06 jun. 2021.

MEHARRAR, A.; TIOURSI, M.; HATTI, M.; STAMBOULI, A. B. A variable speed wind generator maximum power tracking based on adaptative neuro-fuzzy inference system. **Expert Systems with Applications**, v. 38, p. 7659-7664, June 2011, Disponível em: https://doi.org/10.1016/j.eswa.2010.12.163. Acesso em: 06 jun. 2021.

MEHDIZADEH, S.; BEHMANESH, J.; KHALILI, K. Evaluating the performance of artificial intelligence methods for estimation of monthly mean soil temperature without using meteorological data. **Environ Earth Sci**, v. 76, n. 325, Mar. 2017. Disponível em: https://doi.org/10.1007/s12665-017-6607-8. Acesso em: 06 jun. 2021.

PASSINO, K. M.; YURKOVICH, S. **Fuzzy control**. Canada: Addison-Wesley Longman, 1997.

RAJKUMAR, R.; ALBERT, A. J.; CHANDRAKALA, D. Weather Forecasting using Fuzzy Neural Network (FNN) and Hierarchy Particle Swarm Optimization Algorithm (HPSO). **Indian Journal of Science and Technology**, v.8, June 2015. Disponível em: https://indjst.org/download-article.php?Article_Unique_Id=INDJST4094&Full_Text_Pdf_Download=True. Acesso em: 06 jun. 2021.

REDDY, K. J.; SUDHAKAR, N. ANFIS-MPPT control algorithm for a PEMFC system used in electric vehicle applications. **International Journal of Hydrogen Energy**, v. 44, p. 15355-15369, Mar. 2019, Disponível em: https://www-sciencedirect.ez10.periodicos.capes.gov.br/science/article/pii/S0360319919314478?via%3Dihub. Acesso em: 06 jun. 2021.

SHEN, X.; XIE, T.; WANG, T. A Fuzzy Adaptative Backstepping Control Strategy for Marine Current Turbine under Disturbances and Uncertainties. **Energies**, v. 13, n. 24:6550, Dec. 2020. Disponível em: https://doi.org/10.3390/en13246550. Acesso em: 06 jun. 2021.

SHOORANGIZ, R.; MARHABAN, M. H.; Complex Neuro-Fuzzy System for Function Approximation. **International Journal of Applied Electronics in Physics & Robotics**, v. 1, n. 2, p. 5 9, Oct. 2013. Disponível em: http://www.journals.aiac.org.au/index.php/IJAEPR/article/view/190/184. Acesso em: 06 jun. 2021.

SILVA, I. N.; SPATTI, D. H.; FLAUZINO, R. A. **Redes Neurais Artificiais**. 2. ed., São Paulo: Artliber Editora, 2016.

SORAYAEI, A.; SALIMI, M. R.; DIVKOLAII, M. S. Probing efficiency scale of fuzzy neural network on forecasting stock exchange of the automobile industries in Iran. **Indian Journal of Science and Technology**, v. 5, n. 4, Apr. 2012. Disponível em: https://indjst.org/articles/probing-efficiency-scale-of-fuzzy-neural-network-on-forecasting-stock-exchange-of-the-automobile-industries-in-iran. Acesso em: 06 jun. 2021.

TANAKA, M.C. **Utilização da lógica difusa no controle de sistemas eletroidráulicos**. 2011. Dissertação (Mestrado em Engenharia Mecânica) - PPGEM/UFRN, Natal/RN, 2011.

TANAKA, M.C. **Controle inteligente de vibrações utilizando amortecedor magneto reológico**. 2017. Tese (Doutorado em Engenharia Mecânica) - PPGEM/UFRN, Natal/RN, 2017.

XIE, X.; AN, S.; ZHU, C. Fuzzy Neural Network Model and its Application in Water Quality Evaluation. **Nature Environment and Pollution Technology**, v. 15, n. 1, p. 113-116, 2016. Disponível em: https://neptjournal.com/upload-images/NL-55-19-(17)D-278.pdf. Acesso em: 06 jun. 2021.

APÊNDICE A – CÓDIGO EM PYTHON DO ANFIS UTILIZADO

A fim de incentivar trabalhos futuros utilizando o ANFIS, o código utilizado neste trabalho pode ser encontrado abaixo. Caso sofra alterações e melhoras, o código atualizado estará disponível em minha página do GitHub: https://github.com/MasterExecuson/ANFIS/tree/master.

O código abaixo possui funcionalidade apenas para funções com duas variáveis, porém pouco é necessário para estendê-la a três ou mais funções.

```
import pandas as pd
import numpy as np
import collections
import math
import seaborn as sns
sns.set()
from matplotlib import rcParams
rcParams['figure.figsize'] = 5,4
from matplotlib import pyplot
import statsmodels.api as sm
import time
class ANFIS():
  def __init__(self, function, model,Gamma = 10**6,lam = 0.98, delta=1):
    """
      A classe ANFIS define um objeto fixo que será treinado para se encaixar a uma função.
    Para inicializalas precisamos definir o número de funções (function) que corresponde ao número de if rules,
    o formato do modelo que desejamos estudar (model), que deve vir como uma lista com [número de inputs, número de outputs]
    o número máximo de gerações permitido (generations),
    e a taxa de aprendizagem (learningRate).
```

Para treinar a classe e assim obter resultados acurados, é preciso realizar o comando train.

As funções de pertencimento serão gaussianas uniformemente distribuidas, a princípio, sendo modificadas no processo de

backpropagation.

Além do aprendizado do backpropagation, é considerado também o aprendizado por LSE dos parametros de TKS para cada

set de parâmetros das funções de pertencimento que encontrarmos.

Apersar de ser adaptativa, já que seus parâmetros são atualizados confome aprendizado de máquina, ele não atualiza os

parâmetros após a etapa de treinamento.

```
lam: forgetting factor, usually very close to 1. (online learning on LSE algorithm)
self.num_vars = (self.inputdimension+1)*self.function
delta controls the initial state for the LSE learning.
self.A = delta*np.matrix(np.identity(self.num_vars))
"""
self.Gamma = Gamma
self.function = function
self.lam = lam
self.inputdimension = model[0]
self.outputsdimension = model[1]
self.Consqparamsdimension = self.inputdimension+1
self.Wdimension = self.function**self.inputdimension
self.covarienceMatrixDimension = (self.Consqparamsdimension)*(self.function**self.inputdimension)
self.num_vars = self.covarienceMatrixDimension
self.delta = delta
self.Deltamu = (np.zeros([self.inputdimension,self.function]))
self.Deltasig = (np.zeros([self.inputdimension,self.function]))
self.limit = 0.00001
self.E = {-1:1}
def initialize(self,interval=[-10,10],width = 4):
"""
    Initialize the arrays for the needed variables.
```

```
        mu (C) and sig (a) will be defined as an evenly spaced with same lenth argmuents
        """
        self.mu = []
        for eachinput in range(self.inputdimension):
            self.mu.append(np.linspace(interval[0],interval[1],num=self.function))
        self.sig = (np.zeros([self.inputdimension,self.function]) + width)
        self.μ = np.zeros([self.inputdimension,self.function],dtype=np.float64)
    def gaussian(self,x, cont, cont1):



        core = (np.power(x - self.mu[cont][cont1], 2) / (np.power(self.sig[cont][cont1], 2)))
        gauss = np.exp(-core)
        if gauss > self.limit:
            self.delmu[self.iteration][cont][cont1] = 2*(x-self.mu[cont][cont1])*np.exp(-
core)/np.power(self.sig[cont][cont1], 2)
            self.delsig[self.iteration][cont][cont1] = 2*np.power(x-
self.mu[cont][cont1],2)*np.exp(-core)/np.power(self.sig[cont][cont1], 3)


            return gauss
        else:
            self.delmu[self.iteration][cont][cont1] = 0
            self.delsig[self.iteration][cont][cont1] = 0
            return 0
    def membershipFunction(self):
        """
            Apply the inputs to the membership functions to return an array with dimension
(#inputs, #functions).
            This way the μ[0] = [X,Y] will be the membership value for the 0th input and 0th rule.
        """
        cont=0
        for eachinput in self.inputs[self.iteration]: # para cada input
            cont1 = 0
            for eachrule in range(self.function): # para cada função no input
```

```
            self.μ[cont][cont1] = self.gaussian(eachinput,cont,cont1)
            cont1+=1
        cont +=1
    return self.μ
  def Tnorm(self,μ,inputdimension,name, W = None,cont=0,cont2=0): #AND operator
    """
```

Apply the Tnorm function to the μ matrix. Multiplication will be used as "and" operator.

Aqui é feito uma multiplicação de matrizes de n dimensões, sendo n = self.inputdimension (número de inputs).

para isso é necessário que a a dimensão dos vetores μ seja conforme o padrão:

μ[0].shape = [X 1 1 1 ... 1]

μ[1].shape = [1 X 1 1 ... 1]

μ[2].shape = [1 1 X 1 ... 1]

.

.

.

μ[n].shape = [1 1 1 1 ... X]

Sendo X = self.function (número de funções),

e que a multiplicação siga a ordem (1. com 2., 1.2. com 3., 1.2.3. com 4. ...).

isso é feito para realizar a multiplicação matricial das matrizes de multiplas dimensões:

|μ11| |μ11μ21 μ11μ22 μ11μ23|

|μ12| X [μ21 μ22 μ23] = |μ12μ21 μ12μ22 μ12μ23|

|μ13| |μ13μ21 μ13μ22 μ13μ23|

de n dimensões (neste caso duas), cada uma igual ao número de funções para cada variável.

```
    """
    if type(W)==type(None): #primeira passada
        a,cont2 = self.listador(cont2,inputdimension)
        W = μ[cont].reshape(a)
```

```
            cont+=1
            self.__dict__[name] = W
        if cont<inputdimension:
            a,cont2 = self.listador(cont2,inputdimension)

            if cont<inputdimension-1:
                W = np.matmul(μ[cont].reshape(a),W)
            else:
                W = np.matmul(W,μ[cont].reshape(a))
            self.__dict__[name] = W
            cont+=1
            self.__dict__[name] = self.Tnorm(μ,inputdimension,name,W,cont,cont2)
        return self.__dict__[name]
    def listador(self,cont2,inputdimension):
        a = []
        for b in range(inputdimension):
            if b == cont2:
                a.append(self.function)
            else:
                a.append(1)
        cont2+=1
        return a,cont2
    def normalization(self):
        """
        Calculo do W barra, bem fácil com numpy
        """
        self.iterWsum[self.iteration] = np.sum(self.W)
        if self.iterWsum[self.iteration] != 0:
            self.Wbarra = self.W/self.iterWsum[self.iteration]
        else:
            self.Wbarra = self.W
        return self.Wbarra
    def transform(self,Wbarra):
```

""" Makes the transformation of Wbarra from a squared [function**(input)] matrix to a complete matrix for all

the training data, resulting in a matrix of dimension [N] X [(input+1)*(functions)], in order to find the

vector of parameter for the TKS. This is also called as Covarience Matrix.

The final matrix should be:

transformed[0] = [W00*X0,W11*Y0,...,W00,...,Wn0*X0,Wn0*Y0,...,Wn0]

transformed[1] = [W01*X1,W01*Y1,...,W01,...,Wn1*X1,Wn1*Y1,...,Wn1]

.

.

.

transformed[m] = [W0m*Xm,W0m*Ym,...,W0m,...,Wnm*Xm,Wnm*Ym,...,Wnm]

n being the number of inputs and m the nmber of training data pairs.

Wbarra is an list of arrays.

THIS FUNCTION SHOULD BE CALLED AFTER THE NORMALIZATION OFF ****ALL**** THE TRAINING DATA PAIRS HAS BEEN PEFORMED.

A submatrix tem um formato semelhante à Tnorm, para poder calcular a matriz de covariância:

```
[[0.37167694 0.22998752 0.22998752 0.14231245 0.00798868 0.00494326
  0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.        ]

 [0.         0.         0.         0.         0.         0.
  0.37167694 0.22998752 0.22998752 0.14231245 0.00798868 0.00494326
  0.         0.         0.         0.         0.         0.        ]

 [0.         0.         0.         0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.37167694 0.22998752 0.22998752 0.14231245 0.00798868 0.00494326]]
```

```
        for the multiplication:
                |W1 W2 0  0  0  0  |
        [x,y,1] X |0  0  W1 W2 0  0  |
                |0  0  0  0  W1 W2 |
          resultando em uma linha da matriz de covariância para os dados x e y.
      """
      matrixfinal = np.zeros([self.maxiter,self.covarienceMatrixDimension])
      datapaircount = 0
      for matrix in Wbarra: # for each data pair
        submatrix =
np.zeros([self.Consqparamsdimension,self.covarienceMatrixDimension])
        positioncount = 0
        cont1 = 0
        inputcount = 0
        for value in matrix:
          for eachinput in range(self.Consqparamsdimension):
            submatrix[eachinput][positioncount] = value
            positioncount += 1
        temp = self.inputs[datapaircount].copy()
        temp.append(1)
        matrixfinal[datapaircount] = np.matmul((temp),submatrix) # essa multiplicação é uma
forma de alcançar a matrix transformada a partir da matrix de inputs x matriz Wbarra
        datapaircount += 1
      self.covarienceMatrix = matrixfinal
      return self.covarienceMatrix
    def initLSETraining(self):
      #num_vars: number of variables including constant
      #lam: forgetting factor, usually very close to 1 not used on current state.
      self.S = self.Gamma*np.matrix(np.identity(self.num_vars))
      self.A = self.delta*np.matrix(np.identity(self.num_vars))
      self.w = np.matrix(np.zeros(self.num_vars))
      self.w = self.w.reshape(self.w.shape[1],1)
      self.lam_inv = self.lam**(-1)
```

```
        self.sqrt_lam_inv = math.sqrt(self.lam_inv)
        self.a_priori_error = 0
        self.num_obs = 0
    def add_obs(self, x, t):
        '''
        Add the observation x with label t.
        x is a COLUMN vector as a numpy matrix
        t is a real scalar
        '''
        # ((self.S*x)*(x.T*self.S)) me retorna uma matriz cuja diagonal principal é o quadrado dos valores iniciais.
        self.S = self.S - ((self.S*x)*(x.T*self.S))/(1+(x.T*self.S)*x)
        # (t - x.T*self.w) is equivalent to the error before the observation
        self.w = self.w + self.S*x*(t - x.T*self.w)
        self.num_obs += 1


    def fit(self, X, y):
        '''
        Fit a model to X,y.
        X and y are numpy arrays.
        Individual observations in X should have a prepended 1 for constant coefficient.
        '''
        for i in range(len(X)):
            x = np.transpose(np.matrix(X[i]))
            self.add_obs(x,y[i])
    def get_error(self, generation):
        return np.mean(abs(self.E[generation]))
    def predict(self, x):
        '''
        Predict the value of observation x. x should be a numpy matrix (col vector)
        '''
        return float(np.matmul(self.w.T,x))
    def showMembershipFunctions(self, initialgen=None, finalgen=None, step=1):
```

```
        if initialgen == None:
            initialgen = self.generation
        h = lambda x,aa,c: np.exp(-(np.power(x - c, 2) / (np.power(aa, 2))))
        if finalgen == None:
            finalgen = initialgen+1
        for eachinp in range(self.inputdimension):
            for gen in range(initialgen,finalgen,step):
                try:
                    for aa,c in zip(self.gensig[gen][eachinp],self.genmu[gen][eachinp]):
                        l = []
                        for i in range(self.interval[0]*10,self.interval[1]*10,1):
                            l.append(h(i/10,aa,c))
                        pyplot.plot([i/10 for i in range(self.interval[0]*10,self.interval[1]*10,1)],l)
                except:
                    initialgen -= 1
                    for aa,c in zip(self.gensig[gen][eachinp],self.genmu[gen][eachinp]):
                        l = []
                        for i in range(self.interval[0]*10,self.interval[1]*10,1):
                            l.append(h(i/10,aa,c))
                        pyplot.plot([i/10 for i in range(self.interval[0]*10,self.interval[1]*10,1)],l)
            pyplot.show()
    def showErrorHistory(self):
        data = {i:np.mean(abs(self.E[i])) for i in range(self.generation+1)}
        pyplot.plot([i for i in range(self.generation+1)],data.values(),"red",label = "RMSE - Dados de treinamento")
    def LSE(self,w):

        """LSE is the method through which the minimum squared errors of the parameters of the TKS method is calculated"""
        self.pred_y = []
        if self.training:
            self.initLSETraining()
            for i in range(self.maxiter):
```

```
            x = self.covarienceMatrix[i].reshape([self.covarienceMatrixDimension,1])
#transpose
            self.add_obs(np.matrix(x),self.outputs[i])
        else:
            self.w = w
        for i in range(self.maxiter):
            x = self.covarienceMatrix[i].reshape([self.covarienceMatrixDimension,1]) #transpose
            prediction = self.predict(x)
            self.pred_y.append(float(prediction))
        if self.training:
            self.errors = np.array(self.outputs) - self.pred_y
            self.E[self.generation] = (np.array(self.outputs) - self.pred_y)
    def backprop(self):
        """
            Execute the membership function constants tunning via the derivative of the error.
        It's calculated based on the partial derivative of the error for the partial derivative of the
constant.
        As we only know the devivative of the constant regarding the function itself, we need to
apply the chain rule to find
        the partial derivatives of each layer to the next and the derivative of the error withe the
last layer,
        that is easely calculated.
        The derivative calculated is:
        for 2 inputs and 2 rules:
        dEp/dparameter = dEp/dFinalLayer*dGauss/dparameter*[ ((pix + qiy +ri)-
pred_y)*W1121+((pi'x + qi'y +ri')-pred_y)*W1122]/(sumOfWbarra*μOftheparameter),
        It was calculated based on the chain rule on all the layers.
        """
        for eachinput in range(self.inputdimension): # para cada input
            for eachrule in range(self.function): # para cada função nesse input
                newMu = 0
                newSig = 0
                for iteration in range(self.maxiter): #considera cada Datapair existente.
```

```
            count = 0
            newμ = []
            for i in self.iterμ[iteration]:
                if count != eachinput:
                    newμ.append(i)
                else:
                    next
                count += 1
            self.newUsefulW = self.Tnorm(newμ,self.inputdimension-1,"newUsefulW") #
the -1 is needed to remove the counter of the variable we are diving for
            divider = (self.iterWsum[iteration]*self.iterμ[iteration][eachinput][eachrule])
#always positive
            newDivider = (self.iterWsum[iteration]) #always positive
            if divider > self.limit:
                temp = self.inputs[iteration].copy()
                temp.append(1)
                temp = np.array(temp)#,dtype=np.float64)
                self.consequentparameters =
np.array(np.matmul(self.w.reshape([self.function**self.inputdimension,self.Consqparamsdi
mension]),temp),dtype=np.float64)#.reshape([self.function for i in
range(self.inputdimension)])
                self.consequentparameters = self.consequentparameters.reshape([self.function
for i in range(self.inputdimension)])
                idle,usefulParameters,idle = lister(self.iterW[iteration],
self.consequentparameters-self.pred_y[iteration]
                                                    ,eachinput , eachrule,self.inputdimension-1,
                                                    np.zeros(self.function**(self.inputdimension-1)),
                                                    np.zeros(self.function**(self.inputdimension-
1)),self.function)
                newConsequentTerm = np.matmul(self.newUsefulW,usefulParameters)
                newMu +=
-2*(self.errors[iteration])*self.delmu[iteration][eachinput][eachrule]*newConsequentTerm/n
ewDivider
```

```
                newSig += -2*(self.errors[iteration])*self.delsig[iteration][eachinput][eachrule]*newConsequentTerm/newDivider
                self.Deltamu[eachinput][eachrule] = newMu # each variable for each function. should have function * inputs
                self.Deltasig[eachinput][eachrule] = newSig # each variable for each function. should have function * inputs
        n = self.learningRate/abs(np.sum(self.Deltamu)+np.sum(self.Deltasig))
        for eachinput in range(self.inputdimension): # para cada input
            for eachrule in range(self.function): # para cada função nesse input
                self.mu[eachinput][eachrule] += (-1*n)*self.Deltamu[eachinput][eachrule]
                self.sig[eachinput][eachrule] += (-1*n)*self.Deltasig[eachinput][eachrule]
    def train(self, inputs, outputs,interval,width,e=10**-4, retrain=False, generations=500,learningRate=0.98, ResilientProp = True, ResilientPropNred = 2, ResilientPropIncrementValue = 0.01):
        """
        it's expected that inputs is a list with N dimensions and M repetitions of the input, leaving:
        inputs[N,M] as the Nth input on the Mth repetition.
        this way, inputs[0] will return all the variables inputs of the first input.
        Outputs should correspond one to one to the inputs statments.
        """
        self.generations = generations
        self.learningRate = learningRate
        self.ResilientProp = ResilientProp
        self.training = True
        self.outputs = outputs
        self.interval = interval
        self.width = width
        self.ResilientPropIteractionNeeded = ResilientPropNred
        self.ResilientPropIncrementValue = ResilientPropIncrementValue
        self.execution(inputs,e,self.generations)
    def use(self,inputs,generation = None):
```

```
        """
        it's expected that inputs is a list with N dimensions and M repetitions of the input, leaving:
        inputs[N,M] as the Nth input on the Mth repetition.
        If you want to simulate an specific generation use, specify the generation. If passed as none, the best generation will be used.
        """
        if generation == None:
            generation = self.Bestgeneration
        self.training = False
        return self.execution(inputs,10**-28,self.generations+1,generation)
    def test(self, inputs, output):
        E = [np.mean(abs(output-np.array(self.use(inputs,i)))) for i in range(self.generation)]
        self.testE = E
        pyplot.plot( [i for i in range(self.generation)],E,"Blue")
        print("Min Error:",np.min(E))
        f = {E[i]:i for i in range(self.generation)}
        self.Besttesterror = f[np.min(E)]
        print("Min Error Generation:",f[np.min(E)])
    def execution(self,inputs,e,generations,generation = None):
        if not isinstance(inputs,collections.abc.Iterable):
            inputs = [inputs]
        self.inputs = inputs
        self.maxiter = len(self.inputs)
        if self.training:
            reduction = 0
            increase = 0
            self.genw = np.zeros([self.generations,self.num_vars,1])
            dimension = [self.generations,self.maxiter]
            for i in range(self.inputdimension):
                dimension.append(self.function)
            self.genW = np.zeros(dimension)
            self.genmu = np.zeros([self.generations,self.inputdimension,self.function])
            self.gensig = np.zeros([self.generations,self.inputdimension,self.function])
```

```
            self.initialize(self.interval,self.width)
            if len(self.outputs)!=self.maxiter:
                raise Exception(f"Output length ( {self.outputs} ) need to be iqual to input length
( {self.maxiter} )")
            self.generation = 0
            self.Bestgeneration = -1
            error = 0
        end1 = 0
        counter = 3
        while self.generation < generations:
            counter +=1
            ##print("oia")
            start = time.time()
            self.iteration = 0 #controla a iteração do loop de dados de treino de uma geração
            Wbarra = []
            self.iterμ                                                                    =
np.zeros([self.maxiter,self.inputdimension,self.function],dtype=np.float64)
            dimension = [self.maxiter]
            for i in range(self.inputdimension):
                dimension.append(self.function)
            self.iterW = np.zeros(dimension,dtype=np.float64)
            self.iterWsum = np.zeros([self.maxiter]) # valores da soma de todos os W para cada
Datapair
            self.delsig = (np.zeros([self.maxiter,self.inputdimension,self.function]))
            self.delmu = (np.zeros([self.maxiter,self.inputdimension,self.function]))
            if generation != None:
                self.mu = self.genmu[generation]
                self.sig = self.gensig[generation]
            for i in range(self.maxiter):
                μ = (self.membershipFunction())
                self.iterμ[self.iteration] = μ
                self.iterW[self.iteration] = self.Tnorm(self.μ,self.inputdimension,"W")
                Ws = self.normalization().flatten()
```

```
            Wbarra.append(Ws)
            self.iteration+=1
        self.transform(Wbarra)
        self.pred_x = [i for i in range(self.maxiter)]
        self.LSE(w = self.genw[generation])
        if self.training:
            meanerror = self.get_error(self.generation) # np.mean(abs(self.E[self.generation]))
            if meanerror < np.mean(abs(self.E[self.Bestgeneration])) or self.generation == 0:
                self.Bestgeneration = self.generation
            if meanerror < e:
                print("Final Error Square:",meanerror  )
                break
            self.genW[self.generation] = self.iterW
            self.genw[self.generation] = self.w
            self.genmu[self.generation] = self.mu
            self.gensig[self.generation] = self.sig
            end = time.time()
            if counter == 20:
                self.showErrorHistory()
                counter = 0
                print(f"Time on Forward pass:{start - end}")
                print(f"Time on Backward pass:{end1 - start}")
            # change the learning rate based on the signal of the error. One to one approache
            if self.ResilientProp:
                if meanerror < self.get_error(self.generation-1):
                    reduction +=1
                    if reduction >= self.ResilientPropIteractionNeeded:
                        self.learningRate += self.learningRate*self.ResilientPropIncrementValue
#                           print("Increasing learning rate to:",self.learningRate,"...")
                    elif increase == 1:
                        self.learningRate -= self.learningRate*0.1
#                           print("Decreasing learning rate to:",self.learningRate,"..." )
                    increase = 0
```

```
                else:
                    reduction = 0
                    increase = 1
            # change the learning rate based on the signal of the error. Jang approach
#           if self.ResilientProp:
#               if meanerror < self.get_error(self.generation-1):
#                   reduction +=1
#                   if reduction == 4:
#                       self.learningRate += self.learningRate*0.01
#                       print("Increasing learning rate to:",self.learningRate,"...")
#                       reduction = 0
#                   elif increase == 2:
#                        self.learningRate -= self.learningRate*0.1
#                        print("Decreasing learning rate to:",self.learningRate,"..." )
#                        increase = 0
#                   if reduction != 1:
#                       increase = 0
#               else:
#                   reduction = 0
#                   increase += 1
#               print("Generation ",self.generation," mean error:",meanerror)
#           print("Pred_y:", self.pred_y[58])
            if self.generation != self.generations-1:
                end1 = time.time()
                print("==========")
                self.backprop()
        else:
            generations = 0
        self.generation +=1
    self.generation -=1
    if self.training:
        self.showErrorHistory()
        meanerror = self.get_error(self.generation) # np.mean(abs(self.E[self.generation]))
```

```
        print("Final generation ",self.generation," mean error:",meanerror)
    return self.pred_y
def lister(function, function1 ,inp , func, count,summ,summ1, num_func, counter1 = 0, Pass = None):
    """
    Get the terms for the derivative of the third layer to the last, given the input and the function that
    needs to be calculated.
    for the func 1 and inp 1 on a 2 inputs and 2 function basis, we would have:
    summ = [W1121,W1122], summ1 = [(pix + qiy +ri)-pred_y),(pi'x + qi'y +ri')-pred_y)], or vice-versa
    to calculate:
    [ ((pix + qiy +ri)-pred_y)*W1121+((pi'x + qi'y +ri')-pred_y)*W1122], a simple matrix multiplication.
    the parameters sum was already calculated on the consequent parameter variable.
    =
    The first variable will always be the vertical line projection, while the seccond will always be the honrizontal line.
    Thats why the Pass if exists.
    """
    if Pass == None and (inp == 0 or inp == 1):
        inp = (inp**inp)-inp # 0 will turn to 1 and 1 will turn to 0
        Pass = 0
    if type(function) == np.float64:
        summ[counter1] = function
        summ1[counter1] = function1
        counter1 +=1
    elif inp == count:
        count = -1
        summ,summ1,counter1 = lister(function[func],function1[func],inp,func,count,summ,summ1,num_func,counter1,Pass)
    else:
        count-=1
```

```
        for eachvariable in range(num_func):
            summ,summ1,counter1 = lister(function[eachvariable],function1[eachvariable],inp,func,count,summ,summ1,num_func,counter1,Pass)
    return summ,summ1,counter1
```