Trabalho de Conclusão de Curso - Bacharelado em Matemática

Redes Neurais e Algoritmos de Otimização

Thiago Vidal de Abreu

Título: Redes Neurais e Algoritmos de Otimização

Autor: Thiago Vidal de Abreu
Orientador: Prof. Dr. Feodor Pisnitchenko

Banca Examinadora:

Prof. Dr. Jeferson Cassiano
Universidade Federal de do ABC

Prof. Dr.Gustavo Sousa Pavani
Universidade Federal do ABC

Santo André, 30 de novembro de 2022.

Sumário

1 Revisão 10
1.1 **Álgebra Linear** 10
1.1.1 Escalares , Vetores e Matrizes 10
1.1.2 Propriedades de Matrizes 11
1.2 **Otimização irrestrita** 12

2 Redes Neurais 16
2.1 Estrutura de um neurônio 16
2.2 Funções parcialmente diferenciáveis 17
2.3 Funções totalmente diferenciáveis 18
2.4 Arquitetura de redes neurais 19
2.4.1 Arquitetura feedforward de camada simples 19
2.4.2 Arquitetura feedforward de camadas multíplas 20
2.5 Funcionamento da rede 21
2.6 Perceptron 22
2.6.1 Funcionamento do Perceptron 22
2.6.2 Rede Adaline e regra Delta 23
2.6.3 Treinamento de uma rede Perceptron Multicamadas 26
2.6.4 Backpropagation 27
2.6.5 Ajuste de pesos da camada de saída 28
2.6.6 Ajuste de pesos das camadas intermediárias 28

3 Algoritmos para treinamento de uma rede neural 30
3.1 Mínimos Quadrados 30
3.1.1 Mínimos quadrados lineares 31
3.1.2 Mínimos quadrados não lineares 32
3.2 Busca Linear 34
3.2.1 Tamanho do passo 35
3.2.2 Condições de Wolfe 35

3.2.3 Condição de Armijo e Backtracking 38
3.3 Regiões de Confiança 39
3.3.1 Escolhendo o tamanho de Δ_k 39
3.3.2 Ponto de Cauchy 40
3.3.3 Melhorando o ponto de Cauchy - Método Dogleg 43
3.3.4 Levenberg-Marquardt 44

4 Aplicação 47
4.1 Rede neural 47
4.2 Métodos de otimização 47
4.3 Resultados numéricos 48

5 Conclusão 50

Agradecimento

A minha família por todo o apoio e suporte, pois sem eles nada disso seria posível, portanto Newton, Silvia e Gustavo os meus mais sinceros agradecimentos.

À minha namorada Bianca pelo apoio incondicional nessa jornada árdua.

Aos meus amigos Carlos, Ian, Davi, Carla, Eduardo, Luan, Warley por tornarem essa jornada mais leve e ao meu amigo Lucas, que me ajudou muito no decorrer desse Bacharelado em matemática, seja me explicando matéria, discutindo exercicios e até mesmo me animando quando necessário.

Ao meu orientador Fedor Pisnitchenko por me mostrar a beleza da matemática aplicada, pela paciência por me atender em sua sala diversas vezes seja para bater um papo, quanto para me tirar dúvidas de matemática ou programação.

A todos que, de alguma forma, fizeram parte desta importante trajetória da graduação.

Resumo

Uma rede neural consiste de neurônios interconectados que passam informações usando impulsos elétricos e químicos. Os neurônios passam essas informações para outros neurônios e ajustam a informação para realizar uma determinada tarefa. Inspiradas no sistema nervoso biológico,redes neurais artificiais se assemelham ao sistema nervoso biológico, porém não são identicas.

Muito usadas em aplicações como processamento de imagem, processamento de linguagem natural(Natural Language Processing, NLP) e games, as redes neurais artificiais vem ganhando destaque nos dias de hoje.

Apesar do destaque nos dias de hoje, essa teoria bioinspirada já foi estudada desde 1960, porém naquela época não tinhamos o poder computacional suficiente para que uma rede neural artificial pudesse resolver problemas complexos.

Neste trabalho falaremos sobre algumas redes neurais clássicas, suas estruturas e como são treinadas. Além disso falaremos também sobre alguns algoritmos de otimização que podem ser implementados visando treinar uma rede neural.

Palavras Chaves: Redes Neurais, Otimização, Algoritmos

Abstract

A neural network consists of interconnected neurons that pass information using electrical and chemical impulses. Neurons pass on this information to other neurons and adjust the information to perform a certain task. Inspired by the biological nervous system, artificial neural networks resemble the biological nervous system, but are not identical.

Widely used in applications such as image processing, natural language processing (NLP) and games, artificial neural networks are gaining prominence nowadays.

Despite today's prominence, this bioinspired theory has been studied since 1960, but at that time we didn't have enough computational power for an artificial neural network to solve complex problems.

In this work we will talk about some classical neural networks, their structures and how they are trained. In addition, we will also talk about some optimization algorithms that can be implemented to train a neural network.

Keywords: Neural Networks,optimization, algorithms

O primeiro artigo publicado sobre neurocomputação data de 1943, escrito por McCulloch & Pitts. Neste trabalho, os autores criaram o primeiro modelo matemático inspirado no neurônio biológico, fazendo surgir a primeira concepção de neurônio artificial.

Em 1949, Hebb criou o primeiro método de treinamento para redes neurais artificiais, tal método é conhecido como regra de aprendizado de Hebb.

Entre 1957 e 1958, Frank Rosenblatt desenvolveu o primeiro neurocomputador, chamado de *Mark I - Perceptron*.

O modelo do *Perceptron* chamou a atenção da comunidade cientifíca devido à sua capacidade de reconhecer padrões simples. Em 1960 Widrow & Hoff desenvolveram uma rede chamada Adaptative Linear Element, ou *Adaline*, que foi a responsável por promover alguns avanços na área de redes neurais, como por exemplo, o desenvolvimento do algoritmo de aprendizado regra Delta (Gradiente descendente).

As pesquisas nessa área seguiram evoluindo, porém em 1969 com a publicação do livro *Perceptron - an introduction to computational geometry* por Minsky & Papert , a neurocomputação teve suas pesquisas desaceleradas.

Os autores demonstraram a limitação de redes neurais artificiais, constituídas de apenas uma camada (Perceptron e Adaline), em aprender o relacionamento entre as entradas e saídas de funções lógicas simples como o X_{OR}(ou exclusivo lógico).

Mais especifícamente, os autores demonstraram a impossibilidade das redes realizarem a correta classificação de padrões para classes não linearmente separáveis. Após essa publicação pouquissimas pesquisas foram desenvolvidas.

Apenas em 1980 as pesquisas foram retomadas, devido ao desenvolvimento de computadores com maior capacidade de processamento, a criação de algoritmos de otimização mais eficientes e robustos e também as novas descobertas sobre o sistema nervoso biológico. Uma das principais publicações do período foi o livro de Rumelhart, Hinton e Williams, chamado de *Parallel distributed processing* [Rumelhart *et alii*, 1986], em que os autores criaram um algortimo que ajustava os pesos de uma rede que possui mais de uma camada, conseguindo resolver o antigo problema do

X_{OR}, o que reacendeu o interesse das pesquisas nessa área.

Nos dias de hoje, temos várias aplicações práticas de redes neurais artificiais e em diferentes áreas do conhecimento. Podemos destacar o desenvolvimento de algoritmos de aprendizado baseados no método de Levenberg-Marquardt, permitindo aumentar a eficiência do treinamento das redes neurais artificiasem relação ao algoritmo de backpropagation tradicional [Hagan Menha, 1994] [2].

No capítulo 2 descreveremos o que é um neurônio artificial, o que é uma rede neural, como são as estruturas desses objetos e mostraremos algumas redes neurais clássicas e como normalmente treinamos uma rede neural.

No capítulo 3 falaremos sobre algums algoritmos de otimização que podem ser usados para treinar redes neurais, como busca linear, método de regiões de confiança e o método de Levenberg-Marquardt.

1.1 Álgebra Linear

Para o entendimento de redes neurais é preciso conhecer alguns dos principais conceitos de álgebra linear. A seguir introduzimos os conceitos utilizados nesse trabalho.

1.1.1 Escalares , Vetores e Matrizes

- **Corpo**: um corpo $\mathbf{F}$ é um conjunto não vazio , dotado de duas operações binárias $+$ e $\cdot$, denominadas soma e multiplicação por escalar. Nesse trabalho nos restringiremos ao corpo dos números reais.

- **Espaço Vetorial**: Um espaço vetorial $\mathbf{V}$ sobre um corpo $\mathbf{F}$ é um conjunto de elementos chamados vetores, munido de uma operação aditiva
  $+ : \mathbf{V} \times \mathbf{V} \longrightarrow \mathbf{V}$ denominada soma vetorial e também de um produto por escalares $\cdot : \mathbf{F} \times \mathbf{V} \longrightarrow \mathbf{V}$.

- **Base**: uma base de $\mathbf{V}$ é uma lista de vetores em $\mathbf{V}$ que são linearmente independentes e geram todo o espaço $\mathbf{V}$
  Dada uma base, os vetores em $\mathbf{V}$ podem ser representados em função de suas coordenadas e são dados da seguinte maneira:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

  onde x_i se refere a i-ésima coordenada do vetor para $1 \leq i \leq n$.

- **Produto escalar:** Considere $\mathbf{u}$ e $\mathbf{v}$ vetores de $\mathbb{R}^n$ quaisquer, com $\mathbf{u} = [a_1, a_2, \ldots, a_n]^T$ e $\mathbf{v} = [b_1, b_2, \ldots, b_n]^T$. O produto escalar de $\mathbf{u}$ e $\mathbf{v}$ é denotado por $\mathbf{u}^T\mathbf{v}$ e definido

por

$$\mathbf{u}^T\mathbf{v} = a_1b_1 + a_2b_2 + \ldots + a_nb_n \tag{1.1}$$

- **Ortogonalidade de vetores:** Dois vetores $\mathbf{u}$ e $\mathbf{u}$ são ditos ortogonais se

$$\mathbf{u}^T\mathbf{v} = 0$$

- **Norma de um vetor:** A norma euclidiana de um vetor $\mathbf{u} \in \mathbb{R}^n$, denotada por $||\mathbf{u}||$, é definida como a raiz quadrada não negativa de $\mathbf{u}^T\mathbf{u}$. Em particular, se $\mathbf{u} = [a_1, a_2, \ldots, a_n]^T$, então

$$||\mathbf{u}|| = \sqrt{\mathbf{u}^T\mathbf{u}} = \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2}$$

- **Matriz** : Sejam m e n inteiros positivos. Uma matriz m por n é uma lista retangular de elementos de **F**, onde **F** é corpo, com m linhas e n colunas:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}$$

1.1.2 Propriedades de Matrizes

Adição de matrizes: A soma de duas matrizes do mesmo tamanho é uma outra matriz obtida pela soma das entradas de cada uma das matrizes:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \cdots & \vdots \\ b_{m,1} & \cdots & b_{m,n} \end{bmatrix} = \begin{bmatrix} a_{1,1}+b_{1,1} & \cdots & a_{1,n}+b_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1}+b_{m,1} & \cdots & a_{m,n}+b_{m,n} \end{bmatrix}$$

Matriz multiplicada por escalar: O produto de um escalar $\lambda \in \mathbb{R}$ por uma matriz $\mathbf{A} \in \mathbb{R}^{m \times n}$ é uma matriz obtida ao multiplicar cada entrada da matriz pelo escalar:

$$\lambda \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} = \begin{bmatrix} \lambda a_{1,1} & \cdots & \lambda a_{1,n} \\ \vdots & \cdots & \vdots \\ \lambda a_{m,1} & \cdots & \lambda a_{m,n} \end{bmatrix}$$

Multiplicação de Matrizes: Suponha $\mathbf{A} \in \mathbb{R}^{m \times n}$ e $\mathbf{B} \in \mathbb{R}^{n \times p}$. Então definimos $\mathbf{AB} \in$

$\mathbb{R}^{m\times p}$ cuja a entrada da j-ésima linha e k-éisma coluna é dada pela seguinte equação:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} b_{1,1} & \cdots & b_{1,p} \\ \vdots & \cdots & \vdots \\ b_{n,1} & \cdots & b_{n,p} \end{bmatrix} = \begin{bmatrix} c1,1 & \cdots & c_{1,p} \\ \vdots & \cdots & \vdots \\ cm,1 & \cdots & cm,p \end{bmatrix},$$

onde

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j}$$

Observação 1.1 O produto **AB** não está definido se **A** tiver dimensões $m\times p$ e **B** tiver dimensões $q \times n$, com $p \neq q$

Matriz Transposta: A transposta de uma matriz **A**, denotada por, $\mathbf{A^T}$ é a matriz obtida ao trocarmos linhas por colunas. Mais especificamente, se $\mathbf{A} \in \mathbb{R}^{m\times m}$, então $\mathbf{A^T} \in \mathbb{R}^{n\times m}$.

Definição 1.2 ***Matriz positiva definida:***
Seja $A \in \mathbb{R}^{m\times n}$ uma matriz. Dizemos que A é positiva definida quando $x^T Ax > 0$ para todo $x \in \mathbb{R}^n \setminus \{0\}$

1.2 Otimização irrestrita

Em otimização irrestrita, queremos minimizar uma função objetivo que dependa de variáveis reais, onde não existem restrições para os valores dessas variáveis. Sua formulação matemática é da seguinte forma:

$$\min_x f(x),$$

onde $x \in \mathbb{R}^n$ é um vetor real com $n \geq 1$ componentes e $f : \mathbb{R}^n \to \mathbb{R}$ é uma função suave.

Definição 1.3 ***(Função Suave)***
Uma função suave é uma função que possui derivadas contínuas até alguma ordem desejada em algum domínio.

Definição 1.4 *Um ponto $x^* \in \mathbb{R}^n$ é dito mínimo global se $f(x^*) \leq f(x)$ para todo $x \in \mathbb{R}^n$.*

Encontrar um mínimo global não é uma tarefa fácil, pois o nosso conhecimento sobre a função é limitado localmente, ou seja, só conhecemos o seu comportamento em um certo conjunto de pontos e em uma pequena vizinhança ao redor deles. Por causa disso muitos algoritmos são desenvolvidos para encontrar um *mínimo local*, que definiremos a seguir.

Definição 1.5 *Um ponto $x^* \in \mathbb{R}^n$ é um mínimo local se existir uma vizinhança N de x^* tal que $f(x^*) \leq f(x)$ para todo $x \in N$.*

Teorema 1.6 ***(Bolzano-Wierstrass)***
Uma função contínua real f definida em um conjunto fechado e limitado $S \subset \mathbb{R}^n$, admite um minimizador global em S.

Uma das ferramentas mais importante em otimização é a aproximação de funções por Taylor. Nesse trabalho usaremos aproximações de primeira e segunda ordem.
As aproximações de ordem superior, embora sejam mais precisas deixam de ser viáveis pelo alto cuto computacional para o cálculo das derivadas.
Antes de apresentar o teorema de Taylor vamos apresentar conceitos importantes sobre derivadas.
Considere $f : \mathbb{R}^n \to \mathbb{R}$ uma função duas vezes continuamente diferenciável. Indicaremos o gradiente e a Hessiana de f, respectivamente, por

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_N} \end{bmatrix}$$

e

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}. \end{bmatrix}$$

O gradiente de uma função tem propriedades muito interessantes, entre elas temos

- O gradiente é uma direção de crescimento da função;
- é a direção de crescimento mais rápido e
- o gradiente é perpendicular à curva de nível da função.

Teorema 1.7 ***Teorema de Taylor*** *Suponha* $f : \mathbb{R}^n \to \mathbb{R}$ *é continuamente diferenciável e que* $p \in \mathbb{R}^n$. *Assim temos que*

$$f(x+p) = f(x) + \nabla f(x+tp)^T p + r(p), \ \lim_{p \to 0} \frac{r(p)}{||p||} = 0$$

e

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp)^T p + r(p), \ \lim_{p \to 0} \frac{r(p)}{||p||^2} = 0$$

para $t \in (0,1)$.

Definição 1.8 *Um conjunto* $C \subset \mathbb{R}^n$ *é dito convexo quando dados* $x, y \in C$ *o segmento* $[x,y] = \{(1-t)x + ty; t \in [0,1]\}$ *estiver inteiramente contido em* C.

Definição 1.9 *Seja* $C \subset \mathbb{R}^n$ *um conjunto convexo. Dizemos que a função* $f : \mathbb{R}^n \to \mathbb{R}$ *é convexa em* C *quando*

$$f((1-t)x + ty) \leq (1-t)f(x) + tf(y)$$

para todos $x, y \in C$ *e* $t \in [0,1]$.

Teorema 1.10 *Quando* f *é uma função convexa, então qualquer mínimo local* x^* *é um mínimo global de* f.

Teorema 1.11 ***(Condição necessária de 1ª ordem)***
Seja $f : \mathbb{R}^n \to \mathbf{R}$ *diferenciável no ponto* $x^* \in \mathbb{R}^n$. *Se* x^* *é um minimizador local de* f, *então*

$$\nabla f(x^*) = 0.$$

Definição 1.12 *Um ponto* $x^* \in \mathbb{R}^n$ *que cumpre a condição 1.11 é dito ponto crítico ou estácionário da função* f.

Teorema 1.13 ***(Condição necessária de 2ª ordem)***
Seja $f : \mathbb{R}^n \to \mathbf{R}$ *duas vezes difenciável no ponto* $x^* \in \mathbb{R}^n$. *Se* x^* *é um mínimo local de* f, *então a matriz Hessiana de f no ponto* x^* *é semidefinida positiva, ou seja,*

$$d^T \nabla^2 f(x^*) d \geq 0,$$

para todo $d \in \mathbb{R}^n$.

Normalmente não temos muitas informações sobre as funções que queremos minimizar. Tudo que sabemos são os valores de f e talvez algumas das suas derivadas em um certo conjunto de pontos. A partir dessas pequenas informações conseguimos desenvolver algoritmos que consigam identificar soluções aproximadas aos problemas sem um custo computacional tão grande.

2.1 Estrutura de um neurônio

Os neurônios artificiais usados nos modelos de redes neurais artificiais são objetos não-lineares que realizam tarefas simples, eles coletam os dados de entrada, agregam esses dados e produzem uma resposta considerando a função de ativação utilizada. Um exemplo de um neurônio artificial pode ser observado abaixo.

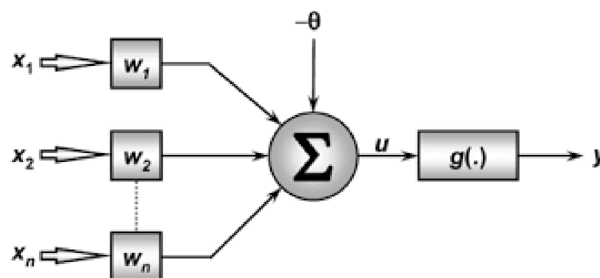

Figura 2.1: Neurônio Artificial.

A partir da imagem acima, conseguimos identificar sete elementos básicos de um neurônio artificial, sendo eles:

- Dados de entrada $[x_1, x_2, ..., x_n]^T$;
- Pesos sinápticos $[w_1, w_2, ..., w_n]^T$, que são os valores utilizados para ponderar a importancia de cada variável de entrada para o resultado final da rede;
- Combinador linear$\{\sum\}$, que seria $\mathbf{w}^T\mathbf{x}$, ou seja, o produto escalar ;
- Limiar de ativação $\{\theta\}$, é uma variável que define qual será o valor que o resultado produzido pelo combinador linear deve atingir para propagar um valor em direção a saída do neurônio;
- Potencial de ativação $\{u\}$, é o resultado obtido pela diferença entre o combinador linear e o limiar de ativação, ou seja

$$u = \mathbf{w}^T\mathbf{x} - \theta$$

(2.1)

- Função de ativação $\{g\}$, o objetivo dessa função é limitar a saída do neurônio dentro de um intervalo de valores assumidos por quem está utilizando a rede neural. Aplicamos g ao potencial de ativação u;
- Dados de saída $\{\mathbf{y}\}$, resultado final produzido pelo neurônio, que pode ser o resultado final , ou pode ser utilizado como resultado intermediário para alimentar um próximo neurônio da rede.

As funções de ativação podem ser divididas em funções parcialmente diferenciáveis e funções totalmente diferenciáveis.

2.2 Funções parcialmente diferenciáveis

As funções de ativação parcialmente diferenciáveis são aquelas que possuem pontos que não tem derivadas.

Principais exemplos desse tipo de função são:

- **Função de Heavyside**

$$g(u) = \begin{cases} 1, \text{ se } u \geq 0 \\ 0, \text{ se } u < 0 \end{cases} \tag{2.2}$$

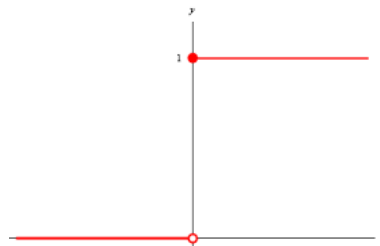

Figura 2.2: Função de Heavyside

- **Função Sinal**

$$g(u) = \begin{cases} 1, \text{ se } u > 0 \\ 0, \text{ se } u = 0 \\ -1, \text{ se } u < 0 \end{cases} \tag{2.3}$$

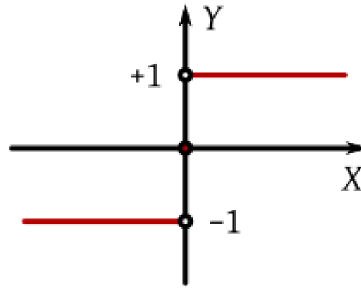

Figura 2.3: Função Sinal

2.3 Funções totalmente diferenciáveis

As funções de ativação totalmente diferenciáveis(contínuas) são aquelas cuja as primeiras derivadas existem para todo ponto do domínio da função.

As principais funções de ativação contínuas são:

- **Função Logística**

$$g(u) = \frac{1}{1 + e^{-\beta u}}, \tag{2.4}$$

onde β é uma constante real associada a inclinção da função logística frente ao seu ponto de inflexão.

Observação 2.1 Quando β for muito grande, a função logística se aproxima da função de Heavyside.

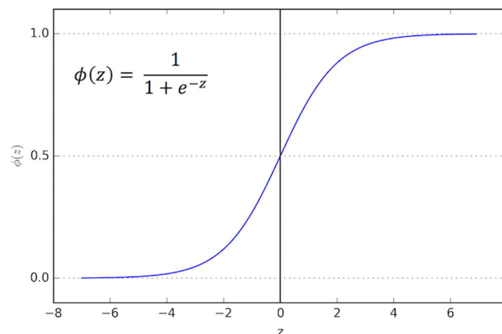

Figura 2.4: Função logística

- **Função tangente hiperbólica**

$$g(u) = \frac{1 - e^{-\beta U}}{1 + e^{-\beta U}} \tag{2.5}$$

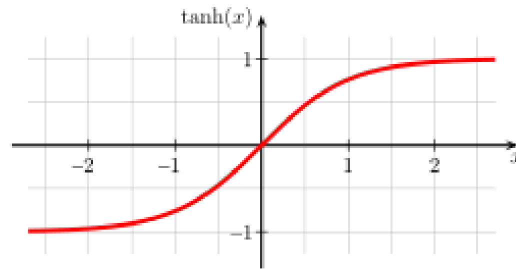

Figura 2.5: Função tangente hiperbólica

Observação 2.2 As funções logística e tangente hiperbólica pertencem à família das funções denominadas *sigmoidais*.

2.4 Arquitetura de redes neurais

A arquitetura de uma rede neural, nada mais é do que a forma como organizamos os neurônios em conjunto, para que estes nos retornem a melhor saída possível. Antes de falarmos das principais arquiteturas de redes neurais, precisamos entender a estrutura de uma primeiro.

Basicamente, dividimos uma rede neural em três partes, também chamadas de camadas, onde cada uma delas pode ter diversos neurônios, sendo elas

- Camada de entrada, que é a camada responsável por receber os dados de entrada. Normalmente esses dados já chegam normalizados para a rede.
- Camadas intermediárias, são aquelas compostas por neurônios que tem a tarefa de extrair caracteristicas associadas aos dados passados como entrada. A mágica acontece nessas camadas, ou seja, o principal processamento feito pelas redes acontece nessa parte.
- Camada de saída, camada responsável por devolver ao usuário um resultado final mediante todo processamento feito pela rede.

Entendido a estrutura de uma rede, vamos agora as suas principais arquiteturas

2.4.1 Arquitetura feedforward de camada simples

Nesse tipo de arquitetura temos apenas duas camadas, sendo elas a camada de entrada e a camada de saída.

A figura abaixo nos mostra um exemplo de arquitetura feedforward de camada simples

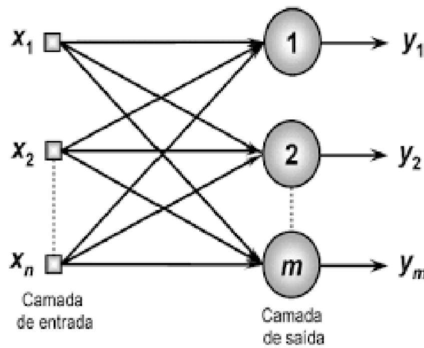

Figura 2.6: Exemplo de rede feedforward de camada simples

Como pode ser observado na imagem acima, o fluxo de informações é unidirecional e o número de dados de saída sempre coincide com o número de neurõnios . Esse tipo de arquitetura é utilizada para resolver problemas lineares, entretanto ela não consegue resolver problemas não-lineares.

Os principais tipos de redes nessa arquitetura são *Perceptron* e *Adaline*, nos quais seus algoritmos de aprendizado são dados pela regra de Hebb e pela regra Delta respectivamente. Falaremos sobre eles mais adiante.

2.4.2 Arquitetura feedforward de camadas multíplas

Nesse tipo de arquitetura já temos as camadas intermediárias. Como essas arquiteturas são mais robustas, normalmente são usadas para realizar tarefas mais complexas como aproximações de funções, previsões de séries temporais e entre outras aplicações.

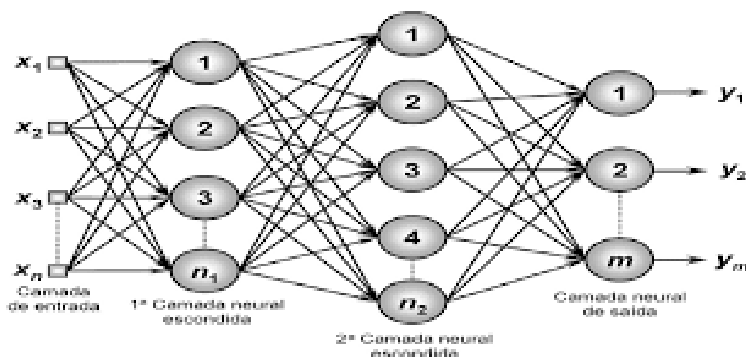

Figura 2.7: Exemplo de rede feedforward de camada simples

Novamente nota-se o fluxo unidirecional de processamento dos dados, assim como o número de dados de saída que coincide com o número de neurônios da respectiva

camada.

Os principais tipos de de redes com arquitetura feedforward de camadas multiplas são *Perceptron multicamadas* e *Radial basis function*(RBF). Os processos de aprendizagem dessas redes são baseados na regra delta generalizada e na regra delta competitiva, que támbem falarameos mais adiante.

2.5 Funcionamento da rede

As redes neurais representam um algoritmo que aprende via exemplos. Basicamente fornecemos à rede certos padrões/caracteristicas e a partir destas a rede aprende relações entre os dados de entrada e de saídas que possam generalizar a tarefa que estamos propondo.

Por exemplo queremos classificar se a imagem passada para a rede é um gato ou não,

- Primeiramente passamos para a rede um conjunto de imagens com gatos e cachorros, considerando como 1 imagens de gatos e 0 imagens de cachorros (esse conjunto de dados normalmente é chamado de dados de treinamento).
- Passados os dados de treino, a rede irá processa-los e buscar por padrões que expliquem as relações entre os dados de entrada(fotos de gatos ou não) e os dados de saída (classificação se a imagem é um gato ou não).
- Depois passamos para a rede neural um outro conjunto de diversas imagens de gatos e cães, mas desta vez sem as respostas(esse conjunto de dados é chamado de dados de teste).
- Com os padrões que a rede aprendeu com os dados de treino, esta irá classificar cada imagem nova do conjunto de teste.
- Depois que todas as imagens de testes foram classificadas, podemos medir a acurácia da rede, ou seja, verificar quantas imagens a rede classificou acertadamente. Se a acurárica estiver baixa, pode ser um indicativo de que precisamos de mais dados para treinar a rede, por outro lado se a rede tem uma acurácia muito alta, pode indicar o fenômeno chamado de *overfitting* que seria como se a rede não aprendesse os padrões, mas sim ”decora-se”as respostas.

Esse tipo de aprendizado descrito acima se chama aprendizado supervisionado.

Agora vamos detalhar , matematicamente como cada uma das arquiteturas citadas anteriormente, de fato são treinadas.

2.6 Perceptron

Uma rede *Perceptron* é constituída por apenas uma camada neural e nessa camada só existe um neurônio artificial.

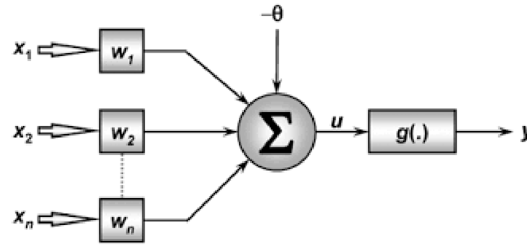

Figura 2.8: Perceptron

2.6.1 Funcionamento do Perceptron

Pela figura 2.8 podemos observar que esse neurônio recebe n dados de entrada. O processamento realizado pelo *Perceptron* pode ser descrito matemáticamente da seguinte forma:

$$\begin{cases} u = \sum_{i=1}^{n} w_i \cdot x_i - \theta \\ \mathbf{y} = g(u) \end{cases} \tag{2.6}$$

em que w_i é o peso associado ao i-ésimo dado de entrada, θ é o limiar de ativação, g é função de ativação e u é o potencial de ativação.

O ajuste dos pesos e limiar do *Perceptron* é realizado pela regra de aprendizado de Hebb [Hebb, 1949]. Resumidamente, se a saída prooduzida pelo *Perceptron* não for a mesma que a passada pelo conjunto de treinamento, então os pesos sinápticos e limiares de ativação da rede serão incrementados proporcionalmente aos dados de entrada. Por outro lado se os dados de saída coincidirem com os dados do conjunto de treinamento, então os pesos e limiares não serão alterados.

Esse processo é repetido para todos os dados de treino até que a saída produzida pela rede seja a mesma que a saída desejada.

Matemáticamente as regras de de ajustes de pesos sinápticos e do limiar de ativação ficam da seguinte maneira:

$$w_i^{atual} = w_i^{anterior} + \eta(d^{(k)} - y)\mathbf{x_i^{(k)}},$$

$$\theta_i^{atual} = \theta_i^{anterior} + \eta(d^{(k)} - y)(-1).$$

Rescrevendo as equações acima na forma vetorial temos:

$$\mathbf{w}^{atual} = \mathbf{w}^{anterior} + \eta(d^{(k)} - y)x_i^{(k)},$$

onde

- $\mathbf{w} = [\theta, w_1, w_2, \cdots, w_n]^T$ é o vetor contendo o limiar de ativação e os pesos,
- $\mathbf{x}^{(k)} = [-1, x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)}]$ é a k-ésima amostra do conjunto de treinamento,
- $d^{(k)}$ é o valor que queremos alcançar para a k-ésima amostra do conjunto de treinamento,
- y é o valor de saída produzido pela rede,
- η é uma constante que define a taxa de aprendizagem da rede.

Sobre a taxa de aprendizagem η ainda não tinhamos comentado, mas ela é a taxa que mostra o quão rápido o processo de treinar a rede vai ocorrer. Normalmente escolhemos $0 \leq \eta \leq 1$.

2.6.2 Rede Adaline e regra Delta

A rede Adaline foi criada por Widrow e Hoff em 1960. Essa rede embora simples, trouxe grandes contribuições para o avanço da área de redes neurais artificiais. A grande contribuição feita, foi a introdução do algoritmo de aprendizado regra Delta, que é o precursor de um dos principais algoritmos de aprendizagem atual.

Assim como o *Perceptron*, o *Adaline* também é constituido de apenas uma camada neural e um neurônio nesta.

O processo de treinamento da rede Adaline é o mesmo processo de treinamento do *Perceptron*, a diferença entre eles ocorre no ajuste dos pesos e no limiar de ativação.

Observe em 2.9 temos um bloco de erro, diferentemente do que temos em 2.8, a função desse erro é ajudar o processo de treinamento da rede, tal erro pode ser descrito por

$$erro = d - u$$

Para treinar uma arquitetura Adaline usamos um algoritmo de aprendizado chamado regra Delta [Widrow & Hoff, 1960], também conhecido como direção de máxima descida ou Gradiente Descendente.

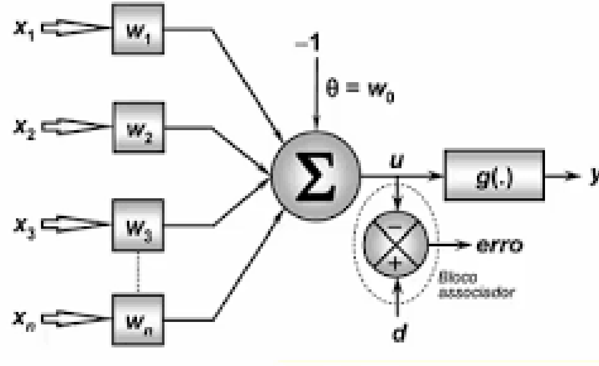

Figura 2.9: Rede *Adaline*

Supondo p amostras de dados de treinamento, a idéia da regra de Delta é minimizar o erro quadrático entre u e d com o intuito de ajustar o vetor de pesos $\mathbf{w} = [\theta, w_1, \ldots, w_n]^T]$.

Assim queremos encontrar um $\mathbf{w}^*$ ótimo tal que o erro quadrático $E(\mathbf{w}^*)$ seja o menor possível, matematicamente temos:

$$E(\mathbf{w}^*) \leq E(\mathbf{w}), \forall \mathbf{w} \in \mathbf{R}^{n+1},$$

onde

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{k=1}^{p}(d^{(k)} - u)^2 & (2.7) \\ &= \frac{1}{2}\sum_{k=1}^{p}(d^{(k)} - (\mathbf{w}^T\mathbf{x}^{(k)} - \theta))^2 & (2.8) \end{aligned}$$

Em seguida vamos aplicar o operador gradiente do erro em relação ao vetor $\mathbf{w}$, ou seja

$$\begin{aligned} \nabla E(\mathbf{w}) &= \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} & (2.9) \\ &= \sum_{k=1}^{p}(d^{(k)} - (\mathbf{w}^T \cdot \mathbf{x}^{(k)} - \theta))(-\mathbf{x}^{(k)}) & (2.10) \\ &= -\sum_{k=1}^{p}(d^{(k)} - u) \cdot (\mathbf{x}^{(k)}) & (2.11) \end{aligned}$$

Como queremos mizimizar o erro quadrático, devemos tomar a direção oposta ao gradiente, sendo assim a variação $\Delta\mathbf{w}$ que deve ser feita no vetor de pesos do *Adaline* é dado por:

$$\Delta\mathbf{w} = -\eta\nabla E(\mathbf{w}) \tag{2.12}$$

que é a mesma coisa que escrevermos

$$\mathbf{w}^{atual} = \mathbf{w}^{anterior} + \eta\sum_{k=1}^{p}(d^{(k)} - u)\cdot(\mathbf{x}^{(k)}) \tag{2.13}$$

Posteriormente comentaremos sobre mínimos quadrados.

2.6.3 Treinamento de uma rede Perceptron Multicamadas

As redes *Perceptron* de mútiplas camadas (PMC) são aquelas que possuem pelo menos uma camada intermediária (escondida) de neurônios, situada entre a camada de entrada e a camada de saída.

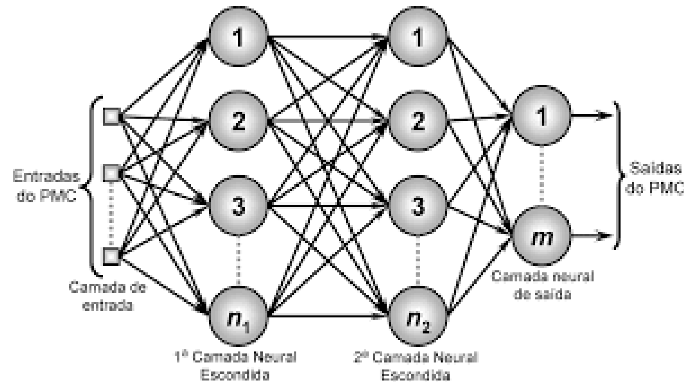

Figura 2.10: Rede *Perceptron* multicamadas

Pela figura 2.10 podemos observar que a saída dos neurônios da primeira camada escondida serve como entrada para os neurônios da segunda camada escondida, que serve como entrada para os neurônios da camada de saída.

Observação 2.3 Normalmente usamos a função *ReLU* dada por

$$g(z) = \max(0, z)$$

como função de ativação nas camadas intermediárias. Tal escolha se deve ao fato da função ReLU adicionar não-linearidade ao modelo da rede.

Diferentemente da rede *Perceptron* e da rede *Adaline*, além de termos camadas intermediárias, também temos que a camada de saída pode ser composta por diversos neurônios, onde cada um destes representaria uma das saídas. Outra diferença observada entre essas redes, é que nas anteriores, um único neurônio era responsável por processar todos os dados de entrada, enquanto que na rede PMC, esse trabalho é distribuído para todos os neurônios da rede.

O treinamento de uma rede Percptron multicamadas ocorre através do algoritmo *backpropagation* que se divide em duas etapas, a primeira o forward propagation, que recebe os dados de treinamento de uma amostra de um conjunto de dados de treino e os propaga camada a camada até a produção das respectivas saídas. Ou seja, nessa fase apenas se realizam os cálculos visando obter as respostas da rede, porém

nesse processo não há ajuste de pesos e limiar. Em seguida as respostas calculadas pela rede são comparadas com as respectivas saídas desejadas, essas comparações são chamadas de erros. Em função desses erros, começamos a segunda etapa do *backpropagation*, chamada de backward. Nessa etapa é que ocorrem os ajustes dos pesos e limiar. A seguir vamos detalhar o *backpropagation*.

2.6.4 Backpropagation

Para entendermos melhor como funciona o *backpropagation*, definiremos a seguir as seguintes variáveis e parâmetros auxiliares.

- $\mathbf{W}_{ji}^{(L)}$ são as matrizes de pesos cujos elementos denotam o valor dos pesos que conectam o j-ésimo neurônio da camada (L) ao i-ésimo neurônio da camada $(L-1)$
- $\mathbf{I}_{j}^{(L)}$ são os vetores cujos elementos denotam a entrada ponderada em relação ao j-ésimo neurônio da camada L , definidos por:

$$\mathbf{I}_{j}^{(L)} = \sum_{i=0}^{n} \mathbf{W}_{ji}^{(L)} \cdot Y_{i}^{(L-1)}$$

- $Y_{j}^{(L)}$ são os vetores cujos elementos denotam a saída do j-ésimo neurônio em relação à camada L, definidos por

$$Y_{j}^{(L)} = \begin{cases} \text{dados de entrada se L = 0} \\ g(I_{j}^{L}) \text{ caso contrário} \end{cases}$$

Considere o exemplo da rede mostrada em 2.10, observe que temos n dados de entrada, n_1 neurônios na primiera camada escondida, n_2 neurônios na segunda camada escondida e n_3 neurônios na camada de saída e o erro dado por

$$E(\mathbf{w}) = \frac{1}{2} \sum_{j=1}^{n_3} (d_{j}^{(k)} - Y_{j}^{(3)})^2, \tag{2.14}$$

onde $Y_{j}^{(3)}$ é o valor produzido pelo j-ésimo neurônio de saída da rede considerando-se a k-ésima amostra dos dados de treino.

2.6.5 Ajuste de pesos da camada de saída

Essa parte é similiar ao que foi feito para treinar a rede *Adaline*, ou seja vamos calcular o gradiente do erro quadrático usando a regra da cadeia, afim de descobrir como o erro varia conforme variamos a matriz de pesos que liga a camada de saída com a segunda camada intermediária.

$$\nabla E^{(3)} = \frac{\partial E}{\partial W_{ji}^{(3)}} = \frac{\partial E}{\partial Y_j^{(3)}} \cdot \frac{\partial Y_j^{(3)}}{\partial I_j^{(3)}} \cdot \frac{\partial I_j^{(3)}}{\partial W_{ji}^{(3)}}, \tag{2.15}$$

onde

$$\frac{\partial I_j^{(3)}}{\partial W_{ji}^{(3)}} = Y_i^2, \tag{2.16}$$

$$\frac{\partial Y_j^{(3)}}{\partial I_j^{(3)}} = g'(I_j^{(3)}), \tag{2.17}$$

e g' é a derivada da função de ativação.

Assim

$$\frac{\partial E}{\partial Y_j^{(3)}} = -(d_j - Y_j^{(3)}). \tag{2.18}$$

Logo

$$\frac{\partial E}{\partial W_{ji}^{(3)}} = -(d_j - Y_j^{(3)}) \cdot g'(I_j^{(3)}) \cdot Y_i^2. \tag{2.19}$$

Assim o ajuste de pesos $W_{ji}^{(3)}$ deve ser feito tomando a direção oposta ao gradiente para minimizarmos o erro, ou seja

$$W_{ji}^{(3)^{atual}} = W_{ji}^{(3)^{anterior}} + \eta \cdot (d_j - Y_j^{(3)}) \cdot g'(I_j^{(3)}) \cdot Y_i^2 \tag{2.20}$$

2.6.6 Ajuste de pesos das camadas intermediárias

Diferentemente dos neurônios da camada de saída, os neurônios das camadas intermediárias não tem um valor desejado dado. Dessa maneira, para fazermos os ajustes nos pesos usamos estimativas de erros de saída produzidos pelos neurônios da camada posterior que já tiveram seus pesos ajustados.

A seguir vamos ajustar os pesos da segunda camada escondida.

$$\nabla E^{(2)} = \frac{\partial E}{\partial W_{ji}^{(2)}} = \frac{\partial E}{\partial Y_j^{(2)}} \cdot \frac{\partial Y_j^{(2)}}{\partial I_j^{(2)}} \cdot \frac{\partial I_j^{(3)}}{\partial W_{ji}^{(2)}} \tag{2.21}$$

onde

$$\frac{\partial I_j^{(2)}}{\partial W_{ji}^{(2)}} = Y_i^1 \tag{2.22}$$

$$\frac{\partial Y_j^{(2)}}{\partial I_j^{(2)}} = g'(I_j^{(2)}), \tag{2.23}$$

e g' é a derivada da função de ativação.

$$\frac{\partial E}{\partial Y_j^{(2)}} = \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^3} \cdot \frac{\partial I_k^3}{\partial W_{kj}^{(3)}} = \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^3} \cdot W_{kj}^{(3)}. \tag{2.24}$$

Novamente tomando a direção oposta ao gradiente temos que a matriz $W_{kj}^{(2)}$ de pesos da segunda camada intermediária é ajustada por

$$W_{ji}^{(2)^{atual}} = W_{ji}^{(2)^{anterior}} + \eta \cdot \sum_{k=1}^{n_3} \frac{\partial E}{\partial I_k^3} \cdot W_{kj}^{(3)} \cdot g'(I_j^{(2)}) \cdot Y_i^1. \tag{2.25}$$

O processo é similar para o ajustar a matriz $W_{kj}^{(1)}$ da primeira camada escondida.

A seguir vamos ver outros possiveis algoritmos de otimização que poderiam ser usados para treinar uma rede neural.

3 Algoritmos para treinamento de uma rede neural

3.1 Mínimos Quadrados

Problemas de mínimos quadrados estão presentes em muitos áreas como em modelos químicos, fisícos, financeiros ou econômicos, muitos deles usam a função definida abaixo para quantificar a diferença entre o modelo proposto e o observado no sistema estudado.

No caso de redes neurais, o problema de mínimos quadrados aparece naturalmente nos treinamentos de vários tipos de redes.

Em problemas de mínimos quadrados a função objetivo é da seguinte forma:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x), \tag{3.1}$$

Ao minimizar $f(x)$, teremos os parâmetros que melhor ajustam o modelo aos dados, onde cada r_j é chamado de residual. Para redes neurais artificiais o residual é chamado de erro e é da saeguinte forma:

$$r_j = (d^k - y), \tag{3.2}$$

onde $d^{(k)}$ é o valor que queremos alcançar para a k-ésima amostra do conjunto de treinamento e y é o valor de saída produzido pela rede.

A seguir vamos entender o motivo pelo qual os minimos quadrados são bastante utilizados.

Seja $r : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, onde

$$r(x) = [r_1(x), r_2(x), \ldots, r_m(x)]^T.$$

Usando essa notação, podemos reescrever f como $f(x) = \frac{1}{2}\|r(x)\|_2^2$. As derivadas de $f(x)$ podem ser escritas em termos do Jacobiano $J(x)$, que é uma matriz de derivada

parciais de primeira ordem dos residuais, definido por

$$J(x) = \left[\frac{\partial r_j}{\partial x_i} \right]_{i=1,2,\ldots,n}^{j=1,2,\ldots,m} = \begin{bmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_m(x)^T \end{bmatrix}$$

onde cada $\nabla r_j(x), j = 1, 2, \cdots, m$ é o gradiente de r_j.

O gradiente e a Hessiana de f podem ser escritas respectivamente da seguinte forma:

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \nabla r_j(x) = J(x)^T r(x) \tag{3.3}$$

$$\nabla^2 f(x) = \sum_{j=1}^{m} \nabla r_j(x) r_j(x)^T + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \tag{3.4}$$

$$= J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \tag{3.5}$$

Em muitas aplicações, as derivadas parciais de primeiro grau , ou seja, a matriz Jacobiana $J(x)$ é relativamente barata de se calcular. Podemos obter o Jacobiano pela fórmula 3.3 , assim podemos calcular o primeiro termo da Hessiana 3.4, sem termos que calcular nenhum derivada parcial de segunda ordem de r_j.

Calcular o primeiro termo da Hessiana sem muito esforço é o que faz com que a família de métodos de mínimos quadrados seja diferenciado dos demais métodos de otimização irrestrita. Além disso, o termo $J(x)^T J(x)$ é frequentemente mais importe do que o segundo termo da Hessiana, já que $\nabla^2 r_x(x)$ ou $r_j(x)$ podem ser pequenos.

3.1.1 Mínimos quadrados lineares

Muitos problemas de ajustes de dados são funções lineares de x. Nesses casos, os resíduos $r_j(x)$ definidos por 3.2 também são lineares, e o problema de minimizar 3.1 é chamado de *mínimios quadrados lineares*.

Note que podemos escrever o residuo como $r_j(\mathbf{x}) = \mathbf{J}\mathbf{x} - \mathbf{y}$ para alguma matrix $\mathbf{J}$ e

algum vetor $\mathbf{y}$ ambos independentes de $\mathbf{x}$, assim a função objetivo é

$$f(x) = \frac{1}{2}\|\mathbf{Jx} - \mathbf{y}\|^2, \tag{3.6}$$

onde $y = r(0)$.

Também temos

$$\nabla f(x) = \mathbf{J}^T(\mathbf{Jx} - \mathbf{y}), \nabla^2 f(x) = \mathbf{J}^T\mathbf{J}. \tag{3.7}$$

É fácil mostrar que 3.6 é convexa , uma propriedade que não necessariamente vale para o problema não linear 3.1. Pelo teorema 3.6 como f é convexa então qualquer ponto $\mathbf{x}^*$ no qual $\nabla f(\mathbf{x}^*) = 0$ é mínimo global. Portanto , $\mathbf{x}^*$ deve satisfazer o seguinte sistema linear:

$$\mathbf{J}^T\mathbf{J}\mathbf{x}* = \mathbf{J}^T\mathbf{y}. \tag{3.8}$$

Abaixo vamos citar três possiveis algoritmos que resolvem o problema de mínimos quadrados linears irrestritos.

- Decomposição Cholesky $\mathbf{J}^T\mathbf{J} = \mathbf{R'}^T\mathbf{R}$
- Decomposição QR
- Decomposição SVD

3.1.2 Mínimos quadrados não lineares

Em um problema de mínimos quadrados não lineares queremos minimizar os resíduos, dados pela seguinte função.

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) \tag{3.9}$$

Para minimizar f, temos duas estratégias:

1. Busca Linear
2. Regiões de confiança

Dado um ponto inicial x_0 o algoritmo de busca linear escolhe uma direção p_k e procura nessa direção a partir do ponto da iteração corrente x_k o próximo ponto x_{k+1} em que o valor da função avaliado em x_{k+1} seja menor do que $f(x_k)$. Para encontrar o quanto o algoritmo deve percorrer na direção p_k deve-se resolver o seguinte problema de minimização.

$$\min_{\alpha>0} f(x_k + \alpha p_k), \tag{3.10}$$

onde α é chamado de tamanho do passo. A solução exata de 3.10 normalmente é cara computacionalmente, para contornar esse problema o algoritmo de busca linear gera uma quantidade limitada the testes de tamanho de passo, até que encontre um que seja próximo do minímo de 3.10. Depois de encontrar o ponto x_{k+1} repete-se o processo em busca de novos pontos, até que o critério de parada do algoritmo seja cumprido.

No método de regiões de confiança, a informação sobre a função objetivo é usada para construir um modelo m_k que visa aproximar f localmente no ponto x_k, ou seja, escolhemos m_k tal que este se comporte de forma similar a f no ponto x_k. Entretanto m_k pode não ser próxima a f quando x estiver longe de x_k, sendo assim restringimos a busca para o mínimo de m_k a uma região em torno de x_k. Em outras palavras, procuramos um candidato ao passo p ao aproximar a solução do seguinte subproblema

$$\min_{p} m_k(x_k + p), \tag{3.11}$$

onde $x_k + p$ está dentro da região de confiança.

Se o candidato à solução não produzir um decrescimento suficiente na função objetivo, então concluímos que a região de confiança é muito grande, sendo assim a diminuímos e re-resolvemos 3.11. Normalmente a região de confiança é uma bola definida por $\|p\|_2 \leq \Delta$, onde Δ é um escalar positivo e é chamado de raio da região de confiança.

Normalmente o modelo m_k é definido como uma função quadrática dada por

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{p^T B_k P}{2}, \tag{3.12}$$

onde ∇f_k e B_k são o gradiente e a Hessiana da função no ponto x_k respectivamente.

Uma diferença entre os algoritmos citados acima é que na busca linear primeiro se escolhe a direção p_k e depois procura-se o tamanho do passo a seguir.

Enquanto que nas regiões de confiança, primeiro se escolhe a distância máxima a se percorrer (raio da região de confiança) e depois procura-se a melhor direção e o passo que melhor diminua a função dada à restrição de distância. Caso esse passo não diminua a função suficientemente, diminuímos o raio de confiança e recalculamos o passo.

A seguir vamos detalhar um pouco mais sobre esses algoritmos de otimização.

3.2 Busca Linear

Busca linear é um método iterativo usado para encontrar o mínimo local de uma função multidimensional não-linear usando o gradiente dessa função. Escolhida uma direção de descida p_k busca-se um tamanho de passo α aceitável que satifaça certas condições.

O método de busca linear pode ser categorizado em dois métodos, exato e não-exato. Nos métodos exatos, busca-se encontrar exatamente o passo que minimiza cada iteração, enquanto que no método não-exato busca-se um tamanho de passo que satisfaça certas condições como a de Wolfe e Goldstein.

Em cada iteração o método de busca linear calcula uma direção p_k e então decide o quanto se mover na mesma. O processo iterativo é dado por

$$x_{k+1} = x_k + \alpha p_k, \tag{3.13}$$

onde α é chamado de tamanho do passo. O sucesso da busca linear depende de boas escolhas da direção p_k e tamanho do passo α_k.
Normalmente a direção p_k escolhida nos métodos de busca linear é a direção de descida, ou seja ,

$$p_k^T \nabla f_k < 0, \tag{3.14}$$

já que está propriedade garante que a função pode ser reduzida ao seguir essa direção.

Normalmente a direção de descida é da forma

$$p_k = -B_k^{-1} \nabla f_k, \tag{3.15}$$

onde B_k é uma matriz simétrica e não singular.

No método de máxima descida, um dos algoritmos mais usado para treinar redes neurais, consideramos B_k como sendo a matriz identidade I. No método de Newton B_k é a Hessiana $\nabla^2 f(x_k)$. Quando p_k é da forma 3.15 e B_k é positiva definida, então

$$p_k^T \nabla f_k = -\nabla f_k B_k^{-1} \nabla f_k < 0, \tag{3.16}$$

e portanto p_j é uma direção de descida.

3.2.1 Tamanho do passo

Ao escolher o tamanho do passo α à ser seguido na direção p_k encontramos um dilema, por um lado queremos que α_k nos forneça uma redução significativa em f, entretanto, não queremos gastar muito tempo fazendo essa escolha.

Para contornar esse problema, normalmente os algoritmos de busca linear elegem alguns candidatos para o valor de α, e quando certas condições forem satisfeitas escolhemos esse candidato para ser o tamanho do passo. Uma das condições que poderia ser imposta, seria pedir que o próximo ponto x_{k+1} reduza f, ou seja, $f(x_{k+1}) = f(x_k + \alpha p_k) < f(x_k)$.

Entretanto como pode ser observado na figura abaixo, essa imposição não é suficiente para garantir convergência. Para evitar tal comportamento, precisamos garantir um condição sufiente de decrescimento, um conceito que falaremos a seguir.

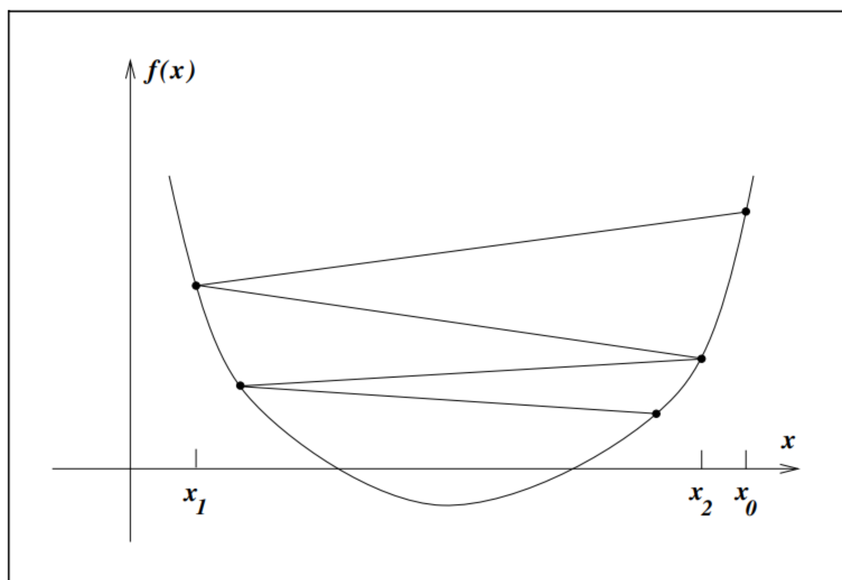

Figura 3.1: Redução insuficiente em f. [1]

3.2.2 Condições de Wolfe

Uma condição muito popular em algoritmos de busca linear não exata, estipula que α_k deve fornecer um decrescimento suficiente da função f, medido por

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \tag{3.17}$$

para alguma constante $c_1 \in (0,1)$. Normalmente escolhemos $c_1 = 10^{-4}$.

A desigualdade 3.17 é chamada de *Condição de Armijo*.

Tomando $f(x_k + \alpha p_k) = \phi(\alpha)$ e $f(x_k) + c_1 \alpha \nabla f_k^T p_k = l(\alpha)$, temos que α será aceito somente se $\phi(\alpha) \leq l(\alpha)$. Podemos ver as regiões em que a condição de Armijo é satisfeita na figura abaixo.

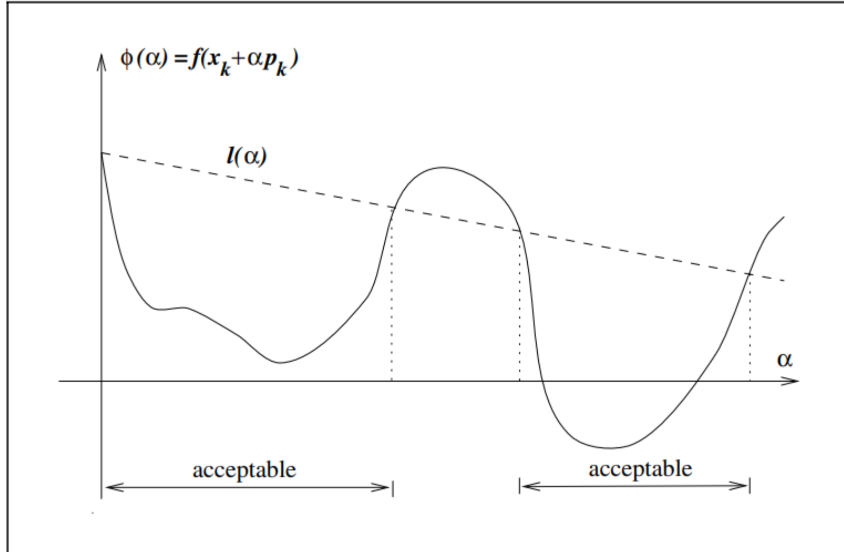

Figura 3.2: Condição de Armijo. [1]

Note que a condição de Armijo não é suficiente para garantir que o algoritmo faça um progresso razoavel, pois como visto na imagem 3.2, para α pequeno a condição já é satisfeita. Para resolver esse problema de passos pequenos temos uma segunda condição chamada de *condição de curvatura*, em que α_k precisa satisfazer

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \tag{3.18}$$

para alguma constante $c_2 \in (c_1, 1)$, onde c_1 é a constante da condição de Armijo.

Essa condição nos diz que a inclinação da função no próximo ponto deve ser maior ou igual à um fração da inclinção do ponto inicial.

Tal condição faz sentido, pois se tivermos uma inclinação inicial fortemente negativa, conseguiremos dar uma passo α_k maior e melhor no sentido de minimização da função.

Por outro lado se a inclinação no ponto inicial não for muito negativa, ou até mesmo positiva, então teremos um forte indicativo de que talvez não possamos diminuir mais a função na direção escolhida, sendo assim faria sentido terminar a busca linear.

Na imagem abaixo podemos ver um exemplo da condição de curvatura.

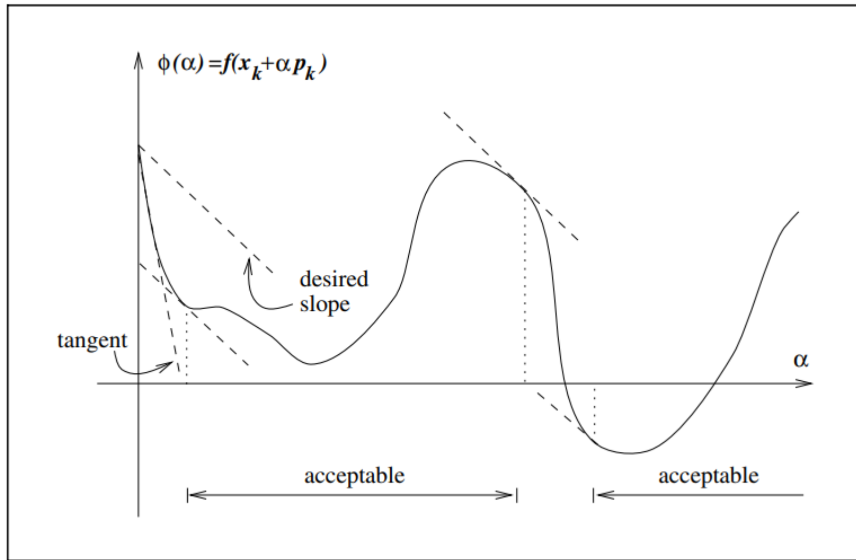

Figura 3.3: Condição de curvatura. [1]

Juntas as condições de Armijo e de curvatura são chamadas de condições de *Wolfe*. Vamos resumi-las a seguir

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \tag{3.19}$$

$$\nabla f(x_k + \alpha_k p_k)^T p_k \geq c_2 \nabla f_k^T p_k, \tag{3.20}$$

com $0 < c_1 < c_2 < 1$. Note que um passo α_k que satisfaça as condições de Wolfe, não necessariamente estará próximo do minímo de ϕ, como pode ser observado na figura abaixo.

Podemos modificar a condição de curvatura para forçar o passo α_k cair em uma vizinhança próxima à um mínimo local de *phi* ou em algum ponto estacionário de ϕ. Essa modificação junto com a condição de Armijo é conhecida como *condição forte de Wolfe* e matematicamente é descrita a seguir.

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k, \tag{3.21}$$

$$|\nabla f(x_k + \alpha_k p_k)^T p_k| \leq c_2 |\nabla f_k^T p_k|, \tag{3.22}$$

com $0 < c_1 < c_2 < 1$.
A condição forte de Wolfe não permite mais que a derivada $\phi'(\alpha_k)$ seja muito positiva, sendo assim ela exclui pontos que estão longe de pontos estacionários de ϕ.

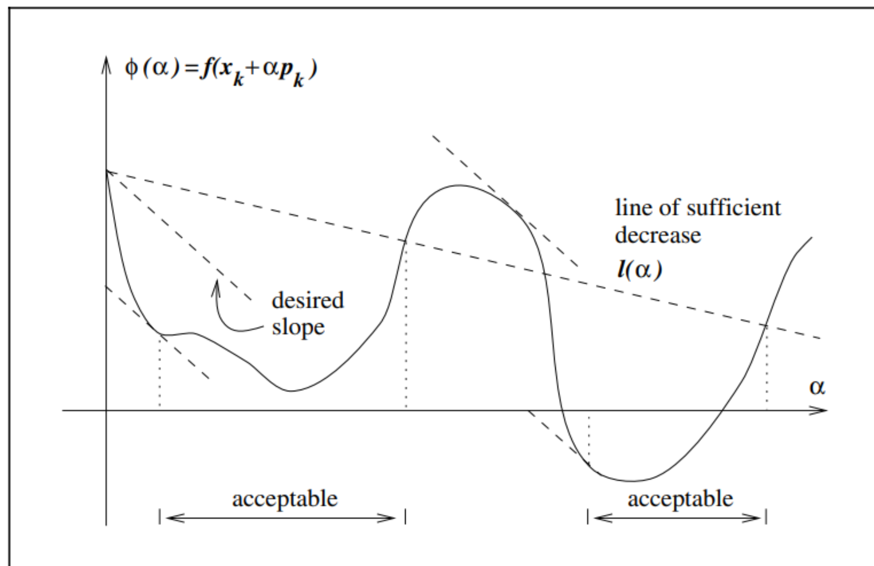

Figura 3.4: Condição de Wolfe. [1]

3.2.3 Condição de Armijo e Backtracking

Um método alternativo ao de Wolfe seria combinar a condição de Armijo com um método chamado de *Backtracking*. Basicamente o que fazemos é verificar se a condição de Armijo é satisfeita, ou seja, se

$$f(x_k + \alpha p_k) \geq f(x_k) + c_1 \alpha \nabla f_k^T p_k,$$

com $c_1 \in (0,1)$. Caso tal condição não seja satisfeita diminuímos α_k, pois sabemos que para pequenos passos pelo menos algum decrescimento na função vamos ter, já que a direção de descida garante pelo menos um pequeno decrescimento em f.

Note que não escolhemos aumentar o tamanho do passo, pois não temos informações sobre a função a partir do passo dado, ou seja ela pode tanto aumentar quanto diminuir.

Em pseudo-código podemos enunciar o *backtracking* da seguinte maneira.

Algorithm 1 Backtracking em busca linear

1: Escolha $\overline{\alpha} > 0$, $\rho \in (0,1)$ e $c \in (0,1)$.
2: Defina $\alpha \leftarrow \overline{\alpha}$
3: **while** $f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k$ **do**
4: $\quad \alpha \leftarrow \rho \alpha$
5: **end while**
6: Termine com $\alpha_k = \alpha$

Nesse algoritmo o tamanho do passo α_k será encontrado após um número finito de iterações, pois eventualmente α_k vai se tornar pequeno o suficiente para satisfazer a condição de Armijo.

3.3 Regiões de Confiança

No método de regiões de confiança, a função objetivo $f(x)$ é aproximada por um modelo $m_k(p)$ em torno de um ponto x_k qualquer. Esse modelo é frequentemente aproximado por uma expansão de Taylor de segunda ordem em torno de x_k, ou seja:

$$f(x+p) \approx m_k(p) = f(x_k) + g_k^T p + \frac{1}{2} p^T H_k p, \tag{3.23}$$

onde $g_k = \nabla f(x)$ é o gradiente do vetor no ponto x_k , $H_k = \nabla^2 f(x_k)$ que é a matriz Hessiana em x_k.

Para encontrarmos o próximo passo p, precisamos minimizar a função modelo $m_k(p)$, mas procuraremos por soluções apenas dentro da região de confiança em que confiamos que $m_k(p)$ seja uma boa aproximação da função objetivo $f(x_k + p)$. Ou seja, procuramos por uma solução no subproblema de região de confiança (TRS)

$$min_{p \in \mathbb{R}^n} m_k(p) = f(x_k) + g_k^T p + \frac{1}{2} p^T H_k p, \tag{3.24}$$

s.t. $\|p\| \leq \Delta_k$, onde $\Delta_k > 0$ é o raio da região de confiança .

3.3.1 Escolhendo o tamanho de Δ_k

Uma métrica muito utilizada para avaliar se o tamanho da região de confiança é adequado, é fazermos o quociente entre a atual redução da função objetivo divida pela redução do modelo quadrático, matematicamente temos

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)} \tag{3.25}$$

Note que como p_k foi obtido minimizando o modelo quadrático m_k no ponto x_k, o denominador de 3.25 sempre será positivo, sendo assim se ρ for negativo teremos que $f(x_k) < f(x_k + p_k)$, ou seja, a função objetivo aumentou, sendo assim rejeitamos o passo p_k.

Por outro lado se ρ_k for próximo de 1, então quer dizer que o modelo m_k é uma boa aproximação da função objetivo, sendo assim podemos expandir o raio de confiança na próxima iteração.

No caso em que ρ for positivo, porém menor que 1, não alteramos o tamanho da região de confiança. Entretanto, se ρ_k for perto de zero ou negativo, então reduzimos o tamanho de Δ_k na próxima iteração.

Algorithm 2 Região de confiança

```
1: Dado D̂ > 0, Δ_0 e η ∈ [0, 1/4)
2: for k = 1, 2, ... do
3:     Obtenha uma aproximação de p_k, resolvendo 3.24
4:     Calcule ρ_k em 3.25
5:     if ρ_k < 1/4 then
6:         Δ_{k+1} = (1/4)Δ_k
7:     else
8:         if ρ_k > 3/4 e ||p_k|| = Δ_k then
9:             Δ_{k+1} = min(2Δ_k, D̂)
10:        else
11:            Δ_{k+1} = Δ_k
12:        end if
13:    end if
14:    if ρ_k > η then
15:        x_{k+1} = x_k + p_k
16:    else
17:        x_{k+1} = x_k
18:    end if
19: end for
```

3.3.2 Ponto de Cauchy

Assim como os métodos de busca linear podem convergir globalmente mesmo não utilizando um tamanho de passo ótimo em cada iteração, também podemos encontrar uma aproximação de tamanho de passo que seja suficiente para garantir a convergência global em métodos de região de confiança.

Esse tamanho de passo aproximado é chamado de passo de *Cauchy* e ele nos fornece uma redução suficiente do modelo m_k que está dentro da região de confiança. Definimos o ponto de Cauchy da seguinte maneira

Algorithm 3 Cálculo do ponto de Cauchy

1: Encontre o vetor p_k^s que resolva a versão linear de 3.24, ou seja,

$$p_k^s = \arg\min_{p \in \mathbb{R}^n} f_k + g_k^T p \tag{3.26}$$

s.t. $\|p\| \leq \Delta_k$

2: Calcule o escalar $\tau_k > 0$ que minimize $m_k(\tau p_k^s)$, satisfazendo o limite da região de confiança, ou seja,

$$\tau_k = \arg\min_{\tau \geq 0} m_k(\tau p_k^s) \tag{3.27}$$

s.t. $\|\tau p_k^s\| \leq \Delta_k$

3: Defina $p_k^c = p_k^s$

Note que a solução de 3.26 é simplesmente

$$p_k^s = -\frac{\Delta_k}{\|g_k\|} g_k,$$

ou seja, é um vetor de tamanho igual a região de confiança orientado na direção de menos gradiente.

Para obtermos τ_k precisamos considerar dois casos separadamente, $g_k^T B_k g_k \leq 0$ e o caso $g_k^T B_k g_k > 0$.

Para o primeiro caso, temos que $m_k(\tau p_k^s)$ decresce monotonicamente com τ sempre que $g_k \neq 0$, sendo assim a solução τ que minimiza o problema 3.27 é igual a 1, pois o mínimo do modelo se encontra na fronteira da região de confiança.

No caso em que $g_k^T B_k g_k$ é positiva definida, para obtermos τ precisamos minimizar $m_k(\tau p_k^s)$ s.a. $\|\tau p_k^s\|$, ou seja, precisamos derivar 3.27 em relação a τ e igualarmos a zero.

Lembrando a equação do modelo quadrático m_k, temos

$$m_k(p) = f_k + g_k^T p + \frac{p^t B_k P}{2}, \tag{3.28}$$

Assim

$$\begin{aligned} m_k(\tau p_k^S) &= f_k + g_k^T \tau p_k^S + \frac{\tau p_k^S B_k \tau p_k^S}{2} \\ &= f_k + g_k^T \tau p_k^S + \frac{\tau^2 p_k^S B_k p_k^S}{2}. \end{aligned} \tag{3.29}$$

Substituindo p_k^S na equação acima, temos

$$\begin{aligned} m_k(\tau p_k^S) &= f_k - \frac{\tau\Delta_k g_k^T g_k}{||g_k||} + \frac{\tau^2\Delta_k^2}{2}\frac{g_k^T B_k g_k}{||g_k||^2} \\ &= f_k - \frac{\tau\Delta_k ||g_k||^2}{||g_k||} + \frac{\tau^2\Delta_k^2}{2}\frac{g_k^T B_k g_k}{||g_k||^2} \\ &= f_k - \tau\Delta_k||g_k|| + \frac{\tau^2\Delta_k^2}{2}\frac{g_k^T B_k g_k}{||g_k||^2} \end{aligned} \tag{3.30}$$

Derivando 3.30 em relação a τ e igualando a zero, temos

$$(m_k(\tau p_k^S))' = -\Delta_k||g_k|| + \tau\Delta_k^2\frac{g_k^T B_k g_k}{||g_k||^2} = 0$$

De onde,

$$\tau = \frac{\Delta_k||g_k||}{\Delta_k^2\frac{g_k^T B_k g_k}{||g_k||^2}}$$

ou

$$\tau = \frac{||g_k||^3}{\Delta_k g_k^T B_k g_k}. \tag{3.31}$$

Assim as possíveis soluções de 3.30 são:

$$\tau_k = \begin{cases} 1, & \text{se } g_k^T B_k g_k \leq 0. \\ \min(1, \frac{||g_k||^3}{\Delta_k g_k^T B_k g_k}), & \text{cc.} \end{cases} \tag{3.32}$$

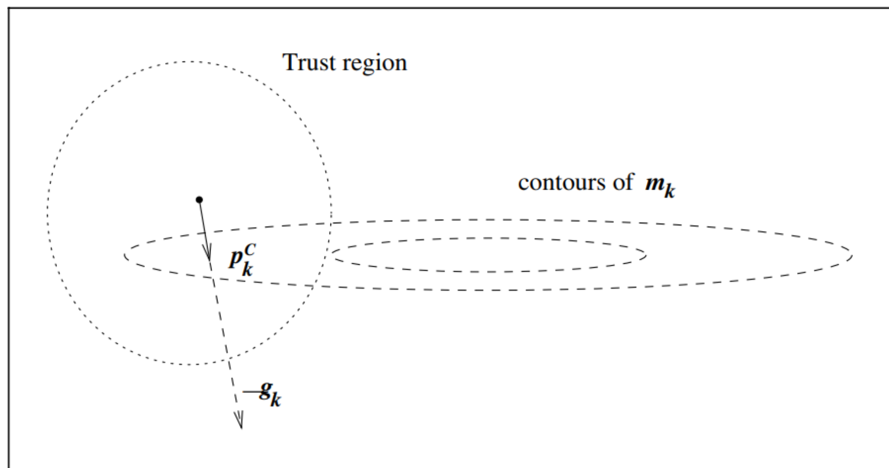

Figura 3.5: Ponto de Cauchy onde B_k é positiva definida. [1]

Apesar do ponto de Cauchy fornecer uma redução suficiente para o modelo m_k, ainda podemos melhorar essa redução, usando outros algoritmos como o método *dogleg* que descreveremos a seguir.

3.3.3 Melhorando o ponto de Cauchy - Método Dogleg

O método Dogleg se aplica quando a Hessiana B_k do modelo quadrático m_k é definida positiva. Este método consiste em minimizar m_k, sujeito à região de confiança, na poligonal que liga os pontos x^k, x_u^k e x_N^k, onde x^k é o ponto atual, x_u^k é o minimizador na direção de menos gradiente e x_N^k é o ponto minimizador irrestrito do modelo, ou seja, o ponto de Newton.

Na figura abaixo ilustramos duas situações. A imagem do lado esquerdo mostra x_u^k dentro da região de confiança, enquanto que a figura à direita mostra x_u^k fora da região de confiança. O ponto obtido pelo método dogleg é representado por x_d^k, enquanto que o ponto minimizador global do modelo dentro da região de confiança é denotado por x_Δ^k.

Abaixo descreveremos o algoritmo do método dogleg.

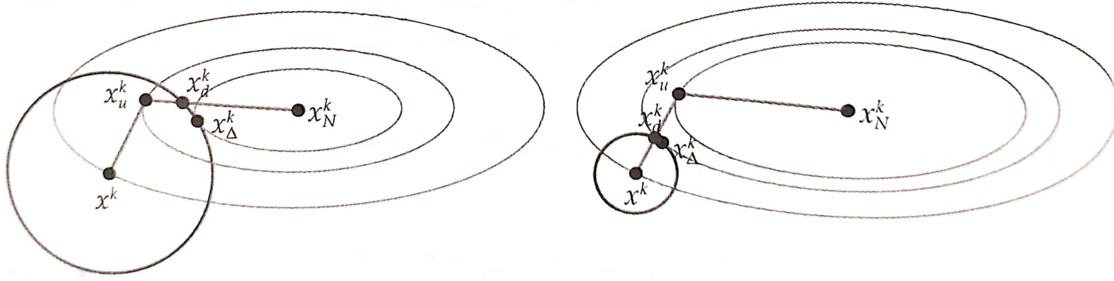

Figura 3.6: Dogleg.[7]

Algorithm 4 Dogleg

1: Dados: $x^k \in \mathbb{R}^n, \Delta_k > 0$
2: Calcule $d_u^k = -\frac{g_k^T g_k}{g_k^T B_k g_k} g_k$
3: **if** $\|d_u^k\| > \Delta_k$ **then**
4: $\quad d^k = -\frac{\Delta_k}{\|g_k\|} g_k$
5: **else**
6: $\quad$ Determine d_N^k tal que $B_k d_N^k = -g_k$
7: $\quad$ **if** $\|d_N^k\| \leq \Delta_k$ **then**
8: $\quad\quad d^k = d_N^k$
9: $\quad$ **else**
10: $\quad\quad$ Determine $\alpha_k \in [0,1]$ tal que $\|d_u^k + \alpha_k(d_N^k - d_u^k)\| = \Delta_k$
11: $\quad$ **end if**
12: **end if**

3.3.4 Levenberg-Marquardt

Como dito anteriormente, o algoritmo *backpropagation* ajusta os valores dos pesos em relação a direção oposta do gradiente da função erro quadrático. Porém esse algoritmo na prática tende a convergir lentamente, exindo muito esforço computacional.

Para contornar esse problema, várias técnicas de otimização tem sido incorporadas ao algoritmo *backprogation* a fim de reduzir o tempo de convergência e o esforço computacional. Uma das técnicas de otimização mais utilizadas para esse propósito destaca-se o algoritmo de Levenberg-Marquardt.

Antes de descrevermos o método de Levenberg-Marquardt, vamos primeiro definir o método de Gauss-Newton (quasi-Newton).

O método de Gauss-Newton é usado para minimizar funções objetivas quadŕáticas como em 3.1, com o intuito de explorar a estrutura do gradiente ∇f em 3.3 e da

hessiana $\nabla^2 f$ em 3.4. Ao invés de resolver as equações de Newton dadas por

$$\nabla^2 f(x_k)p = -\nabla f(x_k), \tag{3.33}$$

fazemos um aproximação de 3.4 removendo o segundo termo $\sum_{j=1}^{m} r_j(x)\nabla^2 r_j(x)$. Assim

$$\nabla^2 f(x_k) \approx J^T J$$

e o problema 3.33 pode ser reescrito como

$$J_k^T J_k p_k^{GN} = -J_k^T r_k, \tag{3.34}$$

onde p_k^{GN} é a direção que estamos procurando.

Uma das vantagens do método de Gauss-Newton é que ele nos abstem do problema de calcular a Hessiana dos resíduos que são necessários como mostrado em 3.4.

Assim como o método de Gauss-Newton usa uma aproximação da Hessiana, o método de Levenberg-Marquardt também o faz.

O algoritmo de Levenberg-Marquardt pode ser descrito e analizado como um problema de regiões de confiança, da mesma forma que descrevemos anteriormente. Sendo assim para uma certa região de confiança, o subproblema a ser resolvido em cada iteração é da forma:

$$min_p \frac{1}{2}\|J_k p + r_k\|^2, \text{ sujeito à } \|p\| \leq \Delta_k, \tag{3.35}$$

onde $\Delta_k > 0$ é a o raio da região de confiança.

Assim estamos escolhendo o modelo $m_k(.)$ 3.24 ser da seguinte forma

$$m_k(p) = \frac{1}{2}\|r_k\|^2 + p^T J_k^T r_k + \frac{1}{2} p^T J_k^T J_k p. \tag{3.36}$$

Pelo o que foi visto sobre regiões de confiança podemos caracterizar a solução do subproblema acima da seguinte forma: Quando a solução do passo de Gauss-Newton p^{GN} estiver dentro da região de confiança(ou seja $\|p^{GN}\| < \Delta$), então esse passo também resolverá o subproblema 3.35. Caso isso não aconteça, então existirá um $\lambda > 0$ tal que a solução de 3.35 será dada por $p = p^{LM}$ que satisfaça $\|p\| = \Delta$ e

$$(J^T J + \lambda I)p = -J^T r \tag{3.37}$$

Assim quando λ for muito pequeno o algoritmo encontrará o passo p_k usando Gauss-Newton, por outro lado se λ for muito grande, note que $J^T J$ não fará muita diferença nessa equação, reduzindo o problema a o algoritmo de máxima descida.

Com o algoritmo de Levenberg-Marquardt conseguimos realizar o treinamento de redes multicamadas na ordem de 10 a 100 vezes mais rápido que o algoritmo de backpropagation convencional [Hagan & Menhaj, 1994].

4 Aplicação

Durante o estudo fizemos a implementação de algumas redes neurais simples e utilizamos algoritmos de otimização distintos, visando à comparação destes com o algoritmo clássico para treinamento de redes neurais, o Gradiente Descendente. A seguir apresentamos os testes com a rede de multiplas camadas. A implementação está disponível pelo link: `https://github.com/thiagovd/nn`.

4.1 Rede neural

A rede implementada possui a seguinte estrutura:

- Camada de entrada: composta por três elementos
- Camada intermediária: composta por dez neurônios
- Camada de saída: compostas por apenas um neurônio

Com essa estrutura temos um total de 40 pesos(váriaveis) à serem encontradas.

A função de ativação utilizada em cada neurónio é a função logística, com coeficiente $\beta = 1$, dada por:

$$g(u) = \frac{1}{1 + e^{-u}} \tag{4.1}$$

O conjunto de dados para treinamento possui duzentas amostras de entrada e duzentas amostras de saída.

4.2 Métodos de otimização

Testamos os seguintes métodos de otimização:

Gradiente Descendente: método clássico descrito no capitulo 2.

Busca Linear: o método BFGS [1]. A particularidade desse método de busca linear está na utilização de uma aproximação para inversa da hessiana e, portanto, não havendo a necessidade de resolver um sistema linear para calcular a direção de descida.

Regiões de confiança (RC): testamos diversos métodos de regiões de confiança. Métodos gerais, aplicados a qualquer problema de minimização irrestrita:

- Exato ([12]): o método classico de regiões de confiança que resolve o subproblema quadratco de forma quase exata.
- Newton-CG (descrito no livro de Nocedal [1], cap. 7.1): utiliza o método de gradiente conjugados para resolver o subproblema quadrático de região de confiança.

Além de algoritmos gerais, testamos os algoritmos espcíficos para resolver o problema de mínimos quadrados (MQ):

- Levenberg–Marquardt: descrito no capítulo 3.
- *Trust Region Reflective* (TRF) ([9] e [10]): método bastante parecido com Levenberg–Marquardt, mas pode ser utilizado com restrições de caixa.
- Dogbox ([11]): método similar ao Dogleg, mas que utiliza região de confiança retangular.

4.3 Resultados numéricos

Para comparar os métodos utilizamos os mesmos critérios de parada para todos os métodos, dados por:

$$\Delta f_k < \epsilon * (\epsilon + f_k)$$

e

$$\|\Delta x_k\| < \epsilon * (\epsilon + \|x_k\|),$$

onde x_k é a variável na iteração k, $f_k = f(x_k)$ é valor da função objetivo na iteração k, $\Delta f_k = f_k - f_{k-1}$ e $\Delta x_k = x_k - x_{k-1}$. Para ϵ escolhemos o valor

$$\epsilon = 10^{-3}.$$

Os resultados com o número de avaliações da função objetivo e Jacobia para cada camada, estão apresentado na tabela a seguir:

Método	Iterações	Erro	f_1	f_2	J_1	J_2
Gradiente Descendente	290	0.07785	580	580	290	290
BFGS	5	0.005455	331	56	331	56
Região de confiança exato	13	0.002876	788	37	595	37
Newton-CG	6	0.002353	152	23	140	23
Levenberg–Marquardt	7	0.002776	151	31	106	25
TRF	8	0.002874	115	24	72	24
Dogbox	20	0.002045	273	54	181	52

Onde f_i é o número de avaliações da função objetivo e J_i é o número de avaliações da Jacobiana, para $i = 1, 2$. Com $i = 1$ referindo-se aos pesos da camada de entrada e para $i = 2$ considerando os pesos da camada de saída.

Observe que o algoritmo clássico, o Gradiente Descendente, performou muito pior que os demais métodos. Foram necessárias um total de 290 iterações para conseguir atingir um erro total de 0.078. Enquanto que o algoritmo Dogbox, o segundo com maior número de iterações, precisou apenas de 20 iterações para chegar a um erro total de 0.002.

Por outro lado, o algoritmo de Levenberg–Marquardt, descrito no capítulo 3, teve uma performance equivalente aos demais algoritmos, exceto Gradiente Descedente. Comparando o número de iterações e erro total,Levenberg–Marquardt ficou com a terceira melhor posição, perdendo apenas para BFGS e Newton-CG no número de iterações e para o Dogbox e Newton-CG no quesito erro total.

Em relação ao número de vezes em que foram necessárias avaliar a função objetivo e a Jacobiana, o algortimo Levenberg–Marquardt também se comportou muito bem, precisando de apenas [151 31] avaliações da função objetivo (f_1 e f_2, respectivamente), enquanto que a média dos demais algoritmos(exceto Gradiente Descendete) foi de [331 38]. Já o número de avaliaçoes da Jacobiana feitas por Levenberg–Marquardt ficou em [106 25], enquanto que a média dos demais algoritmos ficou em [243 38] considerando J_1 e J_2 respectivamente.

5 Conclusão

Nesse trabalho estudamos sobre o que é um neurônio artificial, o que são redes neurais, quais são suas possíveis estruturas, o que significa treinar uma rede neural e como faze-lo.

Aprendemos sobre alguns algoritmos de otimização que podem ser utilizados para treinar as redes neurais, como:

- Gradiente Descendente.
- Levenberg-Marquardt.
- Busca linear.
- Regiões de confiança.
- Dogleg.

Visando à comparação de quais seriam os melhores algoritmos, implementamos diversas redes neurais iguais em estrutura, porém com diferentes algoritmos de otimização. Utilizando o mesmo critério de parada para todas as redes, observamos quais foram os métodos que melhor performaram nos quesitos menor número de iterações e menor erro total.

Dentre os sete algoritmos implementados, observamos que o algortimo clássico para treinar redes neurais, o Gradiente Descendente, não performa tão bem comparado às outras seis implementações.

Enquanto que o algortimo de Levenberg-Marquardt, que foi estudado nesse trabalho não foi o melhor método dentre os sete métodos testados, porém sua performance foi muito similar aos melhores algoritmos.

Referências Bibliográficas

[1] J. Nocedal, S. J. Wright; *Numerical Optimization* Springer;, 1998.

[2] S.D. Spatti and R. Flauzino, *Redes Neurais Artificiais Para Engenharia e Ciências Aplicadas. Fundamentos Teóricos e Aspectos Práticos*, Artliber , 2016.

[3] A. Burkov; *The Hundred-Page Machine Learning Book*, A. Burkov, 2019.

[4] A. Geron; *Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow*, O'REILLY , 2019.

[5] I. Goodfellow, Y. Bengio and A. Courville; *Deep Learning*, MIT Press, 2016.

[6] M. P. Deisenroth, A. A. Faisal and C. S. Ong; *Mathematics for Machine Learning*, Cambridge University Press , 2020.

[7] A.A Riberio and E.W.Karas ; *Otimização contínua, Aspecto teóricos e computacionais*, São Paulo : Cengage Learning, 2013.

[8] J. J. More; *The Levenberg-Marquardt Algorithm: Implementation and Theory*, ed. G. A. Watson, 1977.

[9] M. A. Branch, T. F. Coleman, and Y. Li; *A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems*, SIAM Journal on Scientific Computing, Vol. 21, Number 1, pp 1-23, 1999.

[10] R. H. Byrd, R. B. Schnabel and G. A. Shultz, *Approximate solution of the trust region problem by minimization over two-dimensional subspaces*, Math. Programming, 40, pp. 247-263, 1988.

[11] C. Voglis and I. E. Lagaris ; *A Rectangular Trust Region Dogleg Approach for Unconstrained and Bound Constrained Nonlinear Optimization*, WSEAS International Conference on Applied Mathematics, Corfu, Greece, 2004.

[12] Conn, A. R., Gould, N. I., and Toint, P. L.; *Trust region methods*, Siam. pp. 169-200, 2000.

[13] Hagan, M. T, Menhaj, M. B.; *Training feedforward networks with the Marquardt algorithm*, IEEE Transactions on Neural Networks, vol 5, 1994.