



Sidney Tiago da Silva

Física/UFPR

January 8, 2024

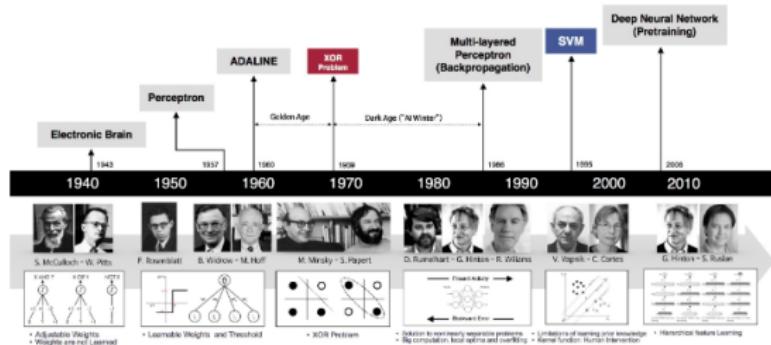
# Contents



# What Is Machine Learning?

A short overview before we jump into  
Deep Learning

# History



Podemos resumir assim os principais marcos na pesquisa e evolução das redes neurais artificiais até chegarmos ao Deep Learning:

**1943:** Warren McCulloch e Walter Pitts criam um modelo computacional para redes neurais baseadas em matemática e algoritmos denominados lógica de limiar.

**1958:** Frank Rosenblatt cria o Perceptron, um algoritmo para o reconhecimento de padrões baseado em uma rede neural computacional de duas camadas usando simples adição e subtração. Ele também propôs camadas adicionais com notações matemáticas, mas isso não seria realizado até 1975.

**1980:** Kunihiko Fukushima propõe a Neoconitron, uma rede neural de hierarquia, multicamada, que foi utilizada para o reconhecimento de caligrafia e outros problemas de reconhecimento de padrões.

**1989:** os cientistas conseguiram criar algoritmos que usavam redes neurais profundas, mas os tempos de treinamento para os sistemas foram medidos em dias, tornando-os impraticáveis para o uso no mundo real.

**1992:** Juyang Weng publica o Cresceptron, um método para realizar o reconhecimento de objetos 3-D automaticamente a partir de cenas desordenadas.

**Meados dos anos 2000:** o termo “aprendizagem profunda” começa a ganhar popularidade após um artigo de Geoffrey Hinton e Ruslan Salakhutdinov mostrar como uma rede neural de várias camadas poderia ser pré-treinada uma camada por vez.

# history

**2009:** acontece o NIPS Workshop sobre Aprendizagem Profunda para Reconhecimento de Voz e descobre-se que com um conjunto de dados suficientemente grande, as redes neurais não precisam de pré-treinamento e as taxas de erro caem significativamente.

**2012:** algoritmos de reconhecimento de padrões artificiais alcançam desempenho em nível humano em determinadas tarefas. E o algoritmo de aprendizagem profunda do Google é capaz de identificar gatos.

**2014:** o Google compra a Startup de Inteligência Artificial chamada DeepMind, do Reino Unido, por £ 400m

**2015:** Facebook coloca a tecnologia de aprendizado profundo – chamada DeepFace – em operação para marcar e identificar automaticamente usuários do Facebook em fotografias. Algoritmos executam tarefas superiores de reconhecimento facial usando redes profundas que levam em conta 120 milhões de parâmetros.

**2016:** o algoritmo do Google DeepMind, AlphaGo, mapeia a arte do complexo jogo de tabuleiro Go e vence o campeão mundial de Go, Lee Sedol, em um torneio altamente divulgado em Seul.

**2017:** adoção em massa do Deep Learning em diversas aplicações corporativas e mobile, além do avanço em pesquisas. Todos os eventos de tecnologia ligados a Data Science, IA e Big Data, apontam Deep Learning como a principal tecnologia para criação de sistemas inteligentes.

## What Is Machine Learning?

“Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed”

— Arthur L. Samuel, AI pioneer, 1959

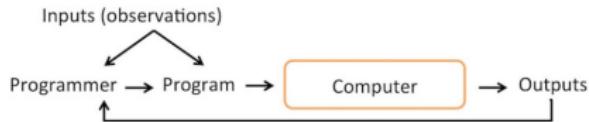
[probably the first, and undoubtedly the most popular definition]

(This is likely not an original quote but a paraphrased version of Samuel's sentence "Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort.")

---

Arthur L. Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

## The Traditional Programming Paradigm

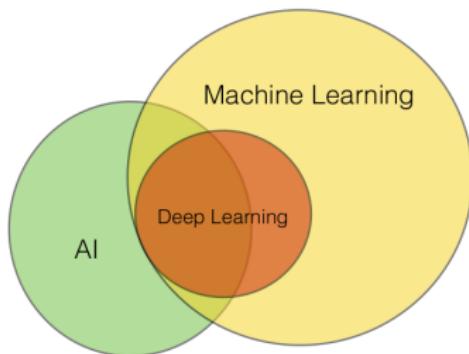


*Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed*  
– Arthur Samuel (1959)

## Machine Learning



## The Connection Between Fields



## Supervised Learning

- Labeled data
- Direct feedback
- Predict outcome/future

## Unsupervised Learning

- No labels/targets
- No feedback
- Find hidden structure in data

## Reinforcement Learning

- Decision process
- Reward system
- Learn series of actions

*Source:* Raschka and Mirjalili (2019). *Python Machine Learning, 3rd Edition*



# Supervised learning

Supervised learning is a subcategory of ML that focuses on learning a classification or regression model from known training data, where we know a priori the specific characteristics of each class, respectively. The principle of this algorithm is to find a “hypothesis” function  $h$  that approximates the unknown function  $f$ , using a classified training data set for this purpose. From this learned function, being able to predict the class of new unknown characteristics.

$$h : \mathcal{X} \longrightarrow \mathcal{Y}, \quad (1)$$

where  $\mathcal{X} = \mathbb{R}^m$  and  $\mathcal{Y} = (1, \dots, k)$  is a vector of integers with class labels  $k$ . In regression, the task of learning a function

$$\mathbb{D} : \mathbb{R}^m \longrightarrow R \quad (2)$$

given the training set

$$\mathbb{D} : \langle x^{[i]}, y^{[i]} \rangle, i = 1, \dots, n, \quad (3)$$

where we denote the  $i$ th training example as  $\langle x^{[i]}, y^{[i]} \rangle$ .

We can represent the training set in the form of a matrix  $\mathbf{X} \in \mathbb{R}^{n \times m}$

$$\mathbf{X} = \begin{pmatrix} x_1^{[1]} & x_2^{[1]} & \dots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \dots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \dots & x_m^{[n]}, \end{pmatrix}$$

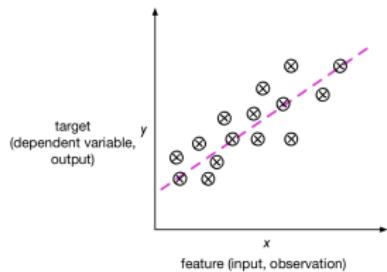
where stores  $n$  training examples and  $m$  the feature number.

Note that to distinguish the characteristic index and the training example index, we will use a superscript notation enclosed in square brackets to refer to the  $i$ th training example and a regular underline notation to refer to the  $j$ th characteristic. The corresponding labels are represented in a column vector  $\mathbf{y}$ ,  $\mathbf{y} \in \mathbb{R}^m$

$$\mathbf{y} = \begin{pmatrix} y^{[1]} \\ y^{[2]} \\ \vdots \\ y^{[n]}. \end{pmatrix} \quad (4)$$

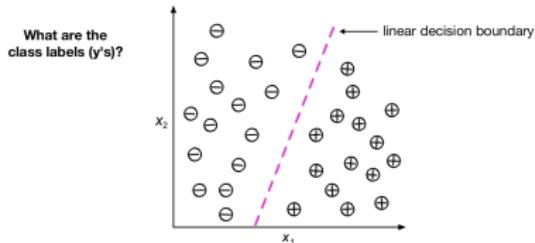
When the values of  $y_i$  are defined by a limited number of discrete values, we have a classification problem.

## Supervised Learning 1: Regression



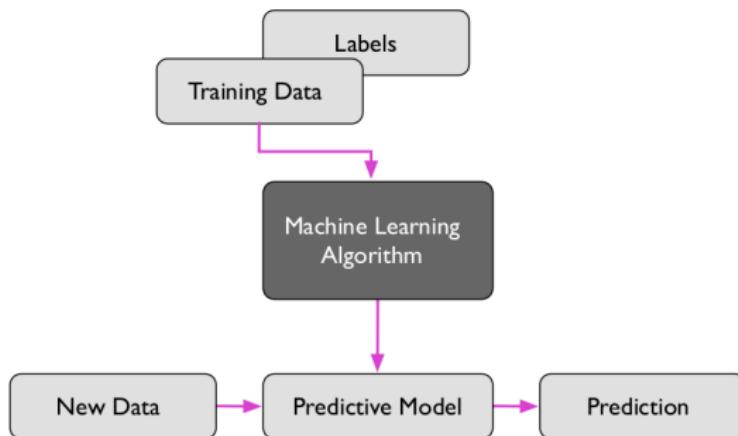
## Supervised Learning 2: Classification

Binary classification example with two features ("independent" variables, predictors)



# Training

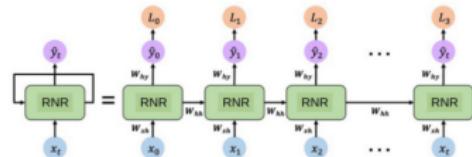
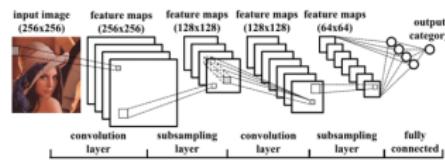
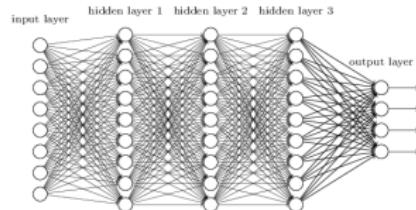
## Supervised Learning Workflow

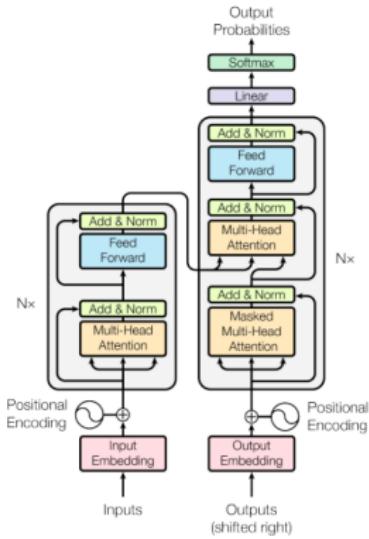
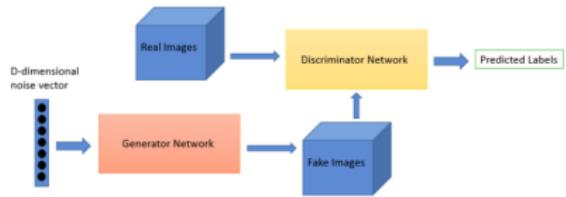




# Deep Learning

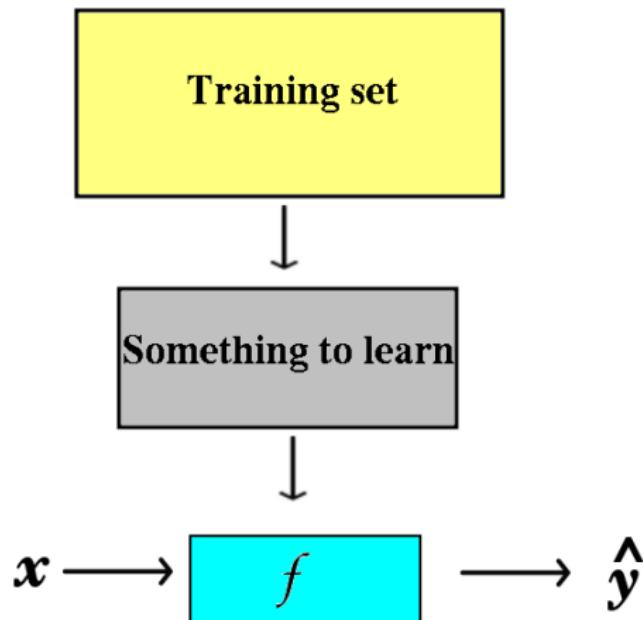
- Multilayer perceptron
- Convolutional networks
  - VGG16
  - ResNet
  - Inception
- Recurring networks
  - LSTM
  - GRU
  - Echo States
- Advanced architectures
  - Transformers
  - GANs – Generative Adversarial Networks



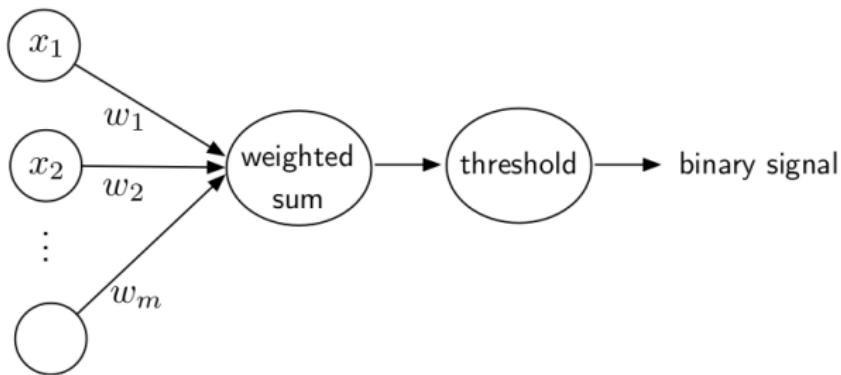




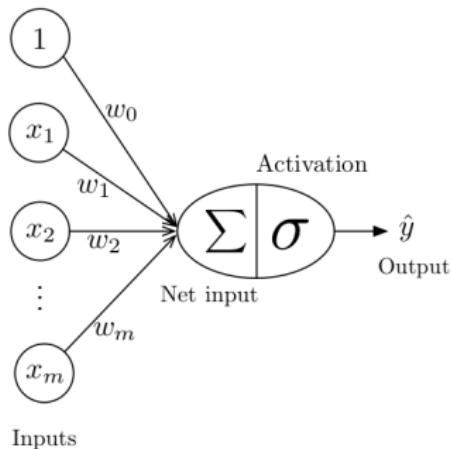
# Perceptron



# Perceptron



# General Notation for Single-Layer Neural Networks



Vector dot product

$$\sigma \left( \sum_{i=0}^m x_i w_i \right) = \sigma(\mathbf{x}^T \mathbf{w}) = \hat{y}$$

$$\sigma(z) = \begin{cases} 0, & z - \theta \leq 0 \\ 1, & z - \theta > 0 \end{cases}$$

$$w_0 = -\theta$$

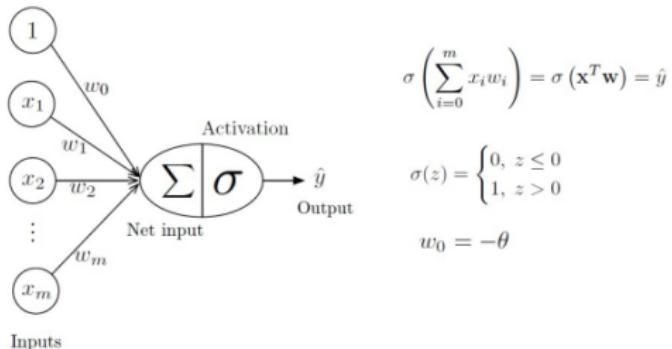
# The Perceptron Learning Algorithm

Let

$$\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \dots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$$

1. Initialize  $\mathbf{w} := 0^m$  (assume notation where weight incl. bias)
2. For every training epoch:
  - A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :
    - (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w})$
    - (b) err :=  $(y^{[i]} - \hat{y}^{[i]})$
    - (c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

# Simulação do Operador Lógico AND



1. Initialize  $\mathbf{w} := 0^m$  (assume notation where weight incl. bias)

2. For every training epoch:

A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$ :

(a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w})$

(b) err :=  $(y^{[i]} - \hat{y}^{[i]})$

(c)  $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$

AND	$x_0$	$x_1$	$x_2$	$t$
Entrada 1:	1	0	0	0
Entrada 2:	1	0	1	0
Entrada 3:	1	1	0	0
Entrada 4:	1	1	1	1

Peso inicial:  $w_0=0, w_1=0, w_2=0$

# 1 epoch

- Apresentar amostra  $x=1,0,0$  and  $y=0$
- Definir a saída da rede  $\hat{y}^{[i]} := \sigma(x^{[i]T} w)$   $\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$ 
  - $z = 1*0+0*0+0*0 = 0$
  - $\hat{y} = \sigma(0) = 0$
- Calcular o erro  $err := (y^{[i]} - \hat{y}^{[i]})$ 
  - $err = 0-0 = 0$
- Atualizar os pesos  $w := w + err \times x^{[i]}$ 
  - $w_0 = 0+0*1 = 0$
  - $w_1 = 0+0*0 = 0$
  - $w_2 = 0+0*0 = 0$

Toda vez que o erro for nulo é equivalente a não fazer nenhuma atualização de pesos!

# 1 epoch

## - Entrada 1

- $\hat{y} = \sigma(1*0+0*0+0*0) = \sigma(0) = 0$
- Err = 0-0 = 0

$$\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w})$$

$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

## - Entrada 2

- $\hat{y} = \sigma(1*0+0*0+1*0) = \sigma(0) = 0$
- Err = 0-0 = 0

$$err := (y^{[i]} - \hat{y}^{[i]})$$

$$\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$$

## - Entrada 3

- $\hat{y} = \sigma(1*0+1*0+0*0) = \sigma(0) = 0$
- Err = 0-0 = 0

## - Entrada 4

- $\hat{y} = \sigma(1*0+1*0+1*0) = \sigma(0) = 0$
- Err = 1-0 = 1
- $w_0 = 0+1*1 = 1$
- $w_1 = 0+1*1 = 1$
- $w_2 = 0+1*1 = 1$

## 2 epoch

### - Entrada 1

- $\hat{y} = \sigma(1*1 + 0*\textcolor{red}{1} + 0*\textcolor{red}{1}) = \sigma(1) = 1$
- Err = 0-1 = -1
- $w_0 = 1 + (-1)*1 = 0$
- $w_1 = 1 + (-1)*0 = 1$
- $w_2 = 1 + (-1)*0 = 1$

$$\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w})$$
$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$
$$\text{err} := (y^{[i]} - \hat{y}^{[i]})$$

### - Entrada 2

- $\hat{y} = \sigma(1*0 + 0*\textcolor{red}{1} + 1*\textcolor{red}{1}) = \sigma(1) = 1$
- Err = 0-1 = -1
- $w_0 = 0 + (-1)*1 = -1$
- $w_1 = 1 + (-1)*0 = 1$
- $w_2 = 1 + (-1)*1 = 0$

$$\mathbf{w} := \mathbf{w} + \text{err} \times \mathbf{x}^{[i]}$$

### - Entrada 3

- $\hat{y} = \sigma(1*(-\textcolor{red}{1}) + 1*\textcolor{red}{1} + 0*\textcolor{red}{0}) = \sigma(0) = 0$
- Err = 0-0 = 0

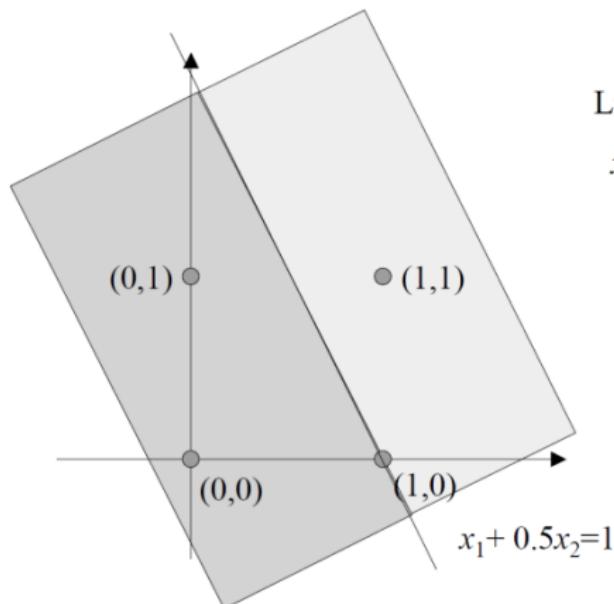
### - Entrada 4

- $\hat{y} = \sigma(1*(-\textcolor{red}{1}) + 1*\textcolor{red}{1} + 1*\textcolor{red}{0}) = \sigma(0) = 0$
- Err = 1-0 = 1
- $w_0 = (-1) + 1*1 = 0$
- $w_1 = 1 + 1*1 = 2$
- $w_2 = 0 + 1*1 = 1$

**Após 5 épocas**

$$w_0 = -2, \quad w_1 = 2, \quad w_2 = 1$$

$$w_0 = -1, \quad w_1 = 1, \quad w_2 = 0,5$$



Linha de Decisão:

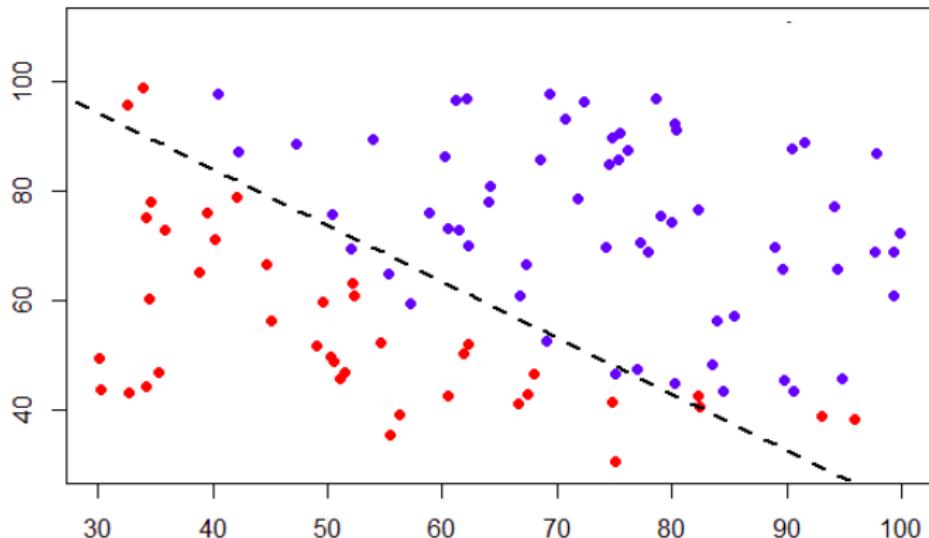
$$x_1 w_1 + x_2 w_2 = -\theta$$



$$x_1 + 0.5 x_2 = 1$$

## ■ Perceptron

- Superfície de decisão linear



- Model

$$\hat{y} = f(\mathbf{x}, \Theta) \quad (5)$$

onde  $\Theta = (\boldsymbol{\omega}, b)$  mapeia a entrada  $\mathbf{x}$  para a saída  $\hat{y}$ .

- $f(\mathbf{x}, \Theta)$  é uma família de funções que relaciona  $\mathbf{x}$  com  $\hat{y}$ . Quando definimos  $\Theta$ , escolhemos um membro dessa família.
- Esses parâmetros são definidos no treinamento.
- $\{x_i, y_i\}_{i=1}^n$ , onde a escolha de  $\Theta$  é feito por meio da função erro  $\mathcal{L}(\Theta)$ , ou seja, o treinamento é a maneira de escolher  $\Theta$  que minimize  $\mathcal{L}(\Theta)$ .

# Função de Perda

- Aprendizagem de máquina é um problema de minimização.
  - Minimizar uma função de custo.

$$\min_{\Theta} \mathcal{L}(\Theta) = \min \mathcal{L}_{\Theta}(y, \hat{y}) \quad (6)$$

- Podemos dizer que o treinamento de uma rede neural é minimizar uma função de custo.

## ■ Propriedade da função custo

- Convexa => Único mínimo global.
- Diferenciável.
- Robustez.
- Suavidade => sem transição ou picos bruscos.
- Monotona => Seu valor diminui a medida que a saída prevista se aproxima da verdadeira.

## ■ Regressão.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n \quad (7)$$

$\hat{y}$  => Valor previsto.

$\beta$  => Pesos.

$x$  => Entradas.

- Função de custo mais utilizadas: Erro quadrático Médio(MSE).  
Erro médio absoluto.  
Função de custo de Huber

## Erro quadrático Médio

$$\mathcal{L}(\Theta) = \frac{1}{2n} \sum_i^n (\hat{y}^i - y^i)^2 \quad (8)$$

A função toma a diferença do valor atual e o valor predito para cada conjunto de treinamento.

$$\mathcal{L}(\Theta) = \frac{1}{2n} \sum_i^n (f(\mathbf{x}^i, \Theta) - y^i)^2 \quad (9)$$

## ■ Propriedade:

- Não negativa.
- Quadrática.
- Diferenciáveis.
- Convexa.
- Dependente da escala
- Não é robusto.

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_i^n |\hat{y}^i - y^i| \quad (10)$$

■ Propriedade:

- Não negativa.
- Linear.
- Robusto.
- Convexa.
- Não é diferenciável.

Função de perda de Huber:

$$\mathcal{L}(\hat{y}, y) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{para } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{para resto.} \end{cases}$$

$\delta$  hiper parâmetro.

- Em deep learning todas as funções de custos são geralmente não convexas, devido as multiplas camadas de funções de ativação não lineares.

## Classificação:

- Binária.
  - Perda binária de entropia cruzada.
- Multi-classe
  - Perda categórica de entropia cruzada.
  - Perda de entropia cruzada esparça.

- Perda binária de entropia cruzada. Maximiza a probabilidade de dada um entrada  $x$  de se opter uma classe  $y$ .

É denotada como sendo:  $p(y = 1|\mathbf{x})$  e  $p(y = 0|\mathbf{x})$ .

Não quero só acertar a classe verdadeira ( $y = 1$ ), mas quero acertar com a maior certeza possível, ou seja, obter a melhor superfície de decisão.

$$\mathcal{L}(y, p(y|\mathbf{x})) = -(y \log(p(y|\mathbf{x})) + (1 - y) \log(1 - p(y|\mathbf{x}))) \quad (11)$$

$$\begin{cases} \log(p) & \text{se } y = 1 \\ \log(1 - p) & \text{se } y = 0. \end{cases}$$

- Perda categórica de entropia cruzada.

$$\mathcal{L} = -\left(\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^c y_{i,j} \log(p_{i,j})\right) \quad (12)$$

função perda é calculada para cada amostra e calculada a media de todo o conjunto de dados.

Para utilizar essa função de perda, temos que transformar as classes em one-hot.

1	0	0
0	1	0
0	0	1

Figure: One-hot para uma tarefa de classificação para três classes.

Função de custo entropia cruzada esparça.

- As classes são codificadas com valores inteiros em vez de vetores one-hot.

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^N \log(\hat{y}_{i,y_i}) \quad (13)$$

onde  $y_i$  é a classe verdadeira e  $\hat{y}_{i,y_i}$  é a probabilidade prevista da i-ésima amostra para a classe correta.

# Gradiente Descendente

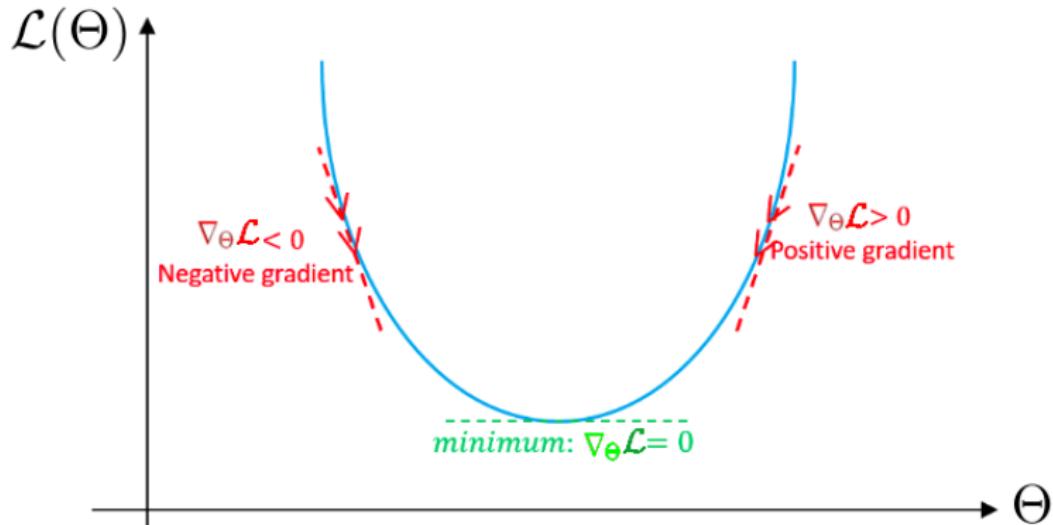
- Queremos obter  $\Theta$  que minimize  $J$ , ou seja,  $\min_{\omega,b} \mathcal{L}(\omega, b)$ 
  - $\Rightarrow$  O mínimo global.
- Podemos determinar o gradiente descendente como:

$$\nabla_{\Theta} \mathcal{L}(\Theta) = \frac{1}{m} \sum_i^n (f(\mathbf{x}^i, \Theta) - y^i) \quad (14)$$

- Algoritmo:

$$\begin{aligned}\omega &=: \omega - \eta \frac{\partial}{\partial \omega} \mathcal{L}(\omega, b) \\ b &=: b - \eta \frac{\partial}{\partial b} \mathcal{L}(\omega, b)\end{aligned} \quad (15)$$

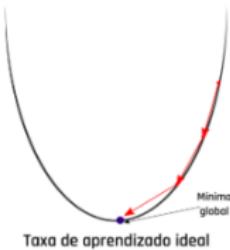
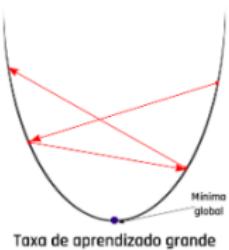
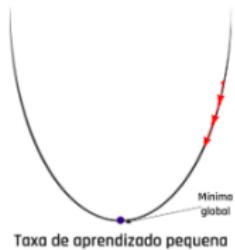
onde  $\eta$  é a taxa de aprendizado.



# Taxa de aprendizado

## ■ Taxa de aprendizado:

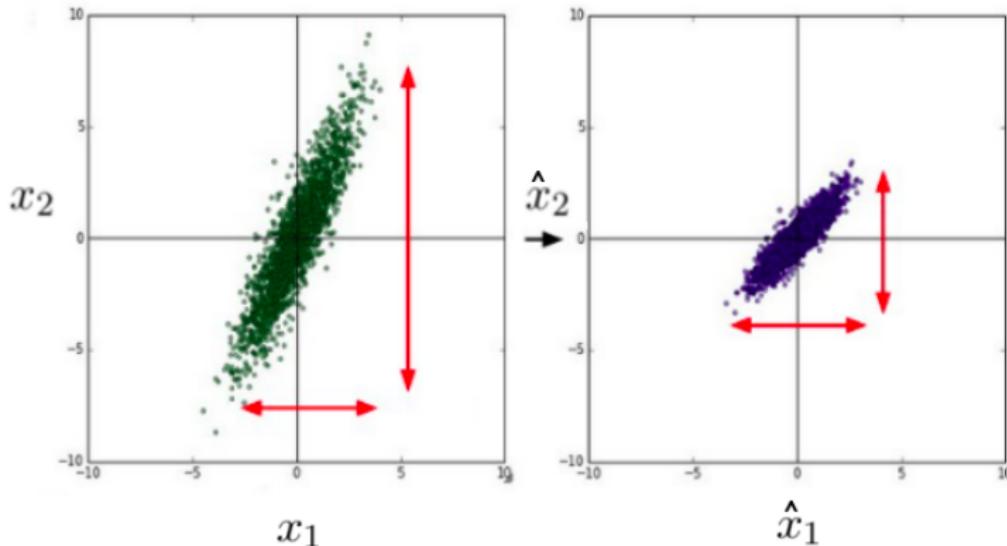
- $\eta$  é muito pequeno, gradiente descendente pode ser muito lento.
- $\eta$  muito grande pode:
  - 1 Nunca alcança o mínimo.
  - 2 Falha para convergir



- Diferença entre gradiente descendente e gradiente descendente estocástico.

Código: Gradiente.

# Normalizando as características



$$f(x) = \omega_1 x_1 + \omega_2 x_2 + b \quad (16)$$

onde  $x_1 \gg x_2$ .

■ Normalização pela média

1

$$\hat{x}_1 = \frac{x_1 - \mu_1}{\max\{x_1\} - \min\{x_1\}}$$

2

$$\hat{x}_2 = \frac{x_2 - \mu_2}{\max\{x_2\} - \min\{x_2\}}$$

onde  $\mu$  é a média.

## ■ Normalização Z-Score

1

$$\hat{x}_1 = \frac{x_1 - \mu_1}{\sigma_1}$$

2

$$\hat{x}_2 = \frac{x_2 - \mu_2}{\sigma_2}$$

onde  $\mu$  é a média e  $\sigma$  é o desvio padrão.

# General Learning Principle

Let  $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \dots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

## "On-line" mode

1. Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$ ,  $b := 0$
2. For every training epoch:
  - A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$  :
    - (a) Compute output (prediction)
    - (b) Calculate error
    - (c) Update  $\mathbf{w}, b$

## Batch mode

1. Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$ ,  $b := 0$
2. For every training epoch:
  - A. Initialize  $\Delta\mathbf{w} := \mathbf{0}$ ,  $\Delta b := 0$
  - B. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$  :
    - (a) Compute output (prediction)
    - (b) Calculate error
    - (c) Update  $\Delta\mathbf{w}, \Delta b$
  - C. Update  $\mathbf{w}, b$  :  
$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, b := +\Delta b$$

# General Learning Principle

Let  $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, \dots, \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0, 1\})^n$

## Minibatch mode

(mix between on-line and batch)

1. Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$ ,  $\mathbf{b} := 0$
2. For every training epoch:
  - A. Initialize  $\Delta\mathbf{w} := 0$ ,  $\Delta b := 0$
  - B. For every  $\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, \dots, \langle \mathbf{x}^{[i+k]}, y^{[i+k]} \rangle\} \subset \mathcal{D}$ :
    - (a) Compute output (prediction)
    - (b) Calculate error
    - (c) Update  $\Delta\mathbf{w}, \Delta b$
  - C. Update  $\mathbf{w}, b$ :  
$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, b := +\Delta b$$

Most commonly used in DL, because

1. Choosing a subset (vs 1 example at a time)  
takes advantage of vectorization (faster iteration through epoch than on-line)
2. having fewer updates than "on-line" makes updates less noisy
3. makes more updates/epoch than "batch" and is thus faster

# (Least-Squares) Linear Regression

The update rule turns out to be this:

"On-line" mode

## Perceptron learning rule

1. Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$ ,  $\mathbf{b} := 0$
2. For every training epoch:

A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$

- (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w} + b)$
- (b)  $\text{err} := (y^{[i]} - \hat{y}^{[i]})$
- (c)  $\mathbf{w} := \mathbf{w} + \text{err} \times \mathbf{x}^{[i]}$   
 $b := b + \text{err}$

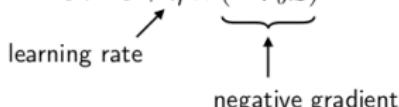
## Stochastic gradient descent

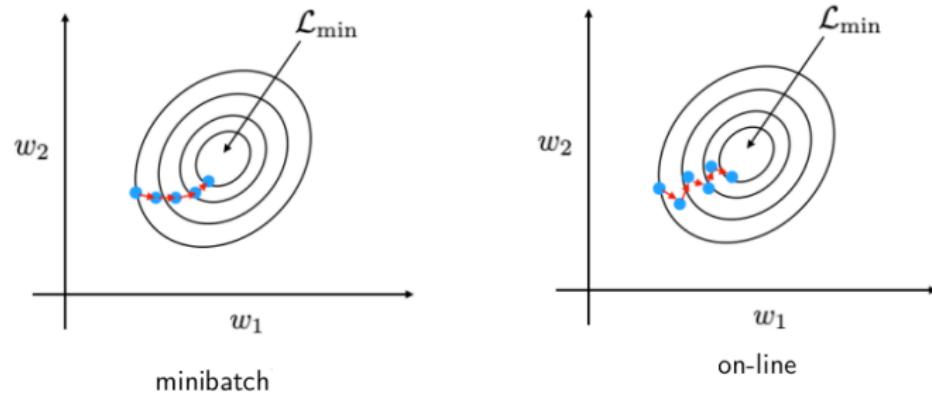
1. Initialize  $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$ ,  $\mathbf{b} := 0$
2. For every training epoch:

A. For every  $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$

- (a)  $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T} \mathbf{w} + b)$
- (b)  $\nabla_{\mathbf{w}} \mathcal{L} = (y^{[i]} - \hat{y}^{[i]}) \mathbf{x}^{[i]}$   
 $\nabla_b \mathcal{L} = (y^{[i]} - \hat{y}^{[i]})$

- (c)  $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}} \mathcal{L})$   
 $b := b + \eta \times (-\nabla_b \mathcal{L})$

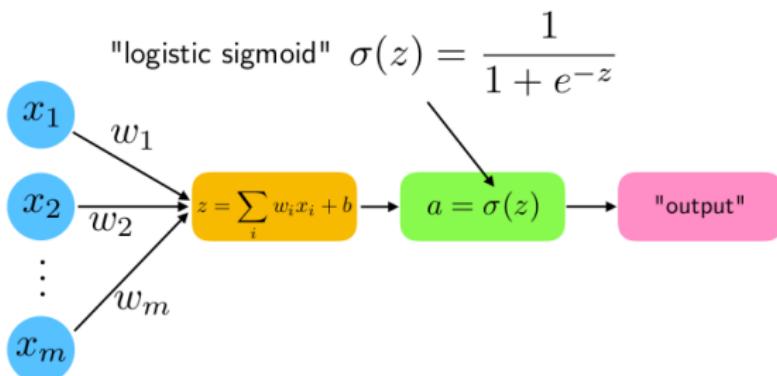




# Multi-class Classification

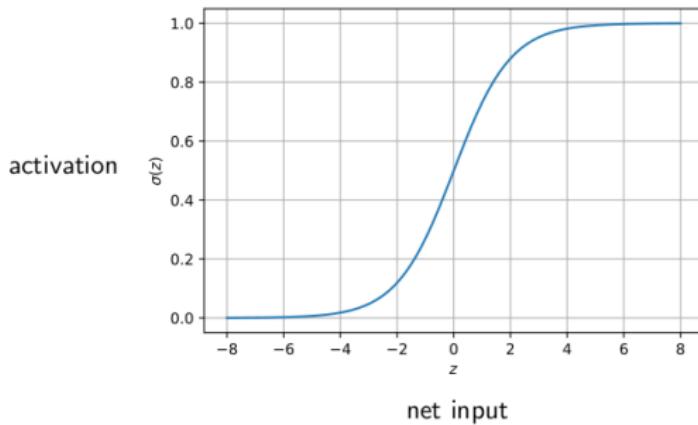
## Logistic Regression Neuron

For binary classes  $y \in \{0, 1\}$



# Logistic Sigmoid Function

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



## Log-Likelihood "Loss"

$$\begin{aligned}L(\mathbf{w}) &= P(\mathbf{y} \mid \mathbf{x}; \mathbf{w}) \\&= \prod_{i=1}^n P(y^{(i)} \mid x^{(i)}; \mathbf{w}) \\&= \prod_{i=1}^n \left( \sigma(z^{(i)}) \right)^{y^{(i)}} \left( 1 - \sigma(z^{(i)}) \right)^{1-y^{(i)}}\end{aligned}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$\begin{aligned}l(\mathbf{w}) &= \log L(\mathbf{w}) \\&= \sum_{i=1}^n [y^{(i)} \log (\sigma(z^{(i)})) + (1 - y^{(i)}) \log (1 - \sigma(z^{(i)}))]\end{aligned}$$

## Negative Log-Likelihood Loss

In practice, it is even more convenient to minimize negative log-likelihood instead of maximizing log-likelihood:

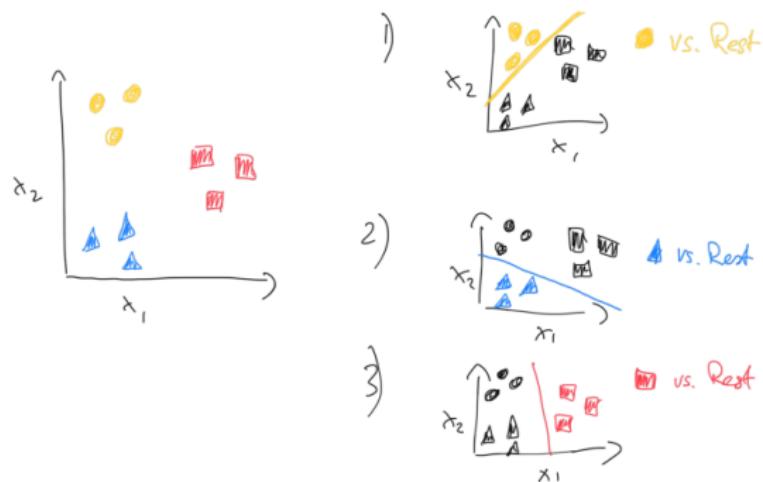
$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= -l(\mathbf{w}) \\ &= - \sum_{i=1}^n [y^{(i)} \log (\sigma(z^{(i)})) + (1 - y^{(i)}) \log (1 - \sigma(z^{(i)}))]\end{aligned}$$

(in code, we also usually add a  $1/n$  scaling factor for further convenience, where  $n$  is the number of training examples or number of examples in a minibatch)

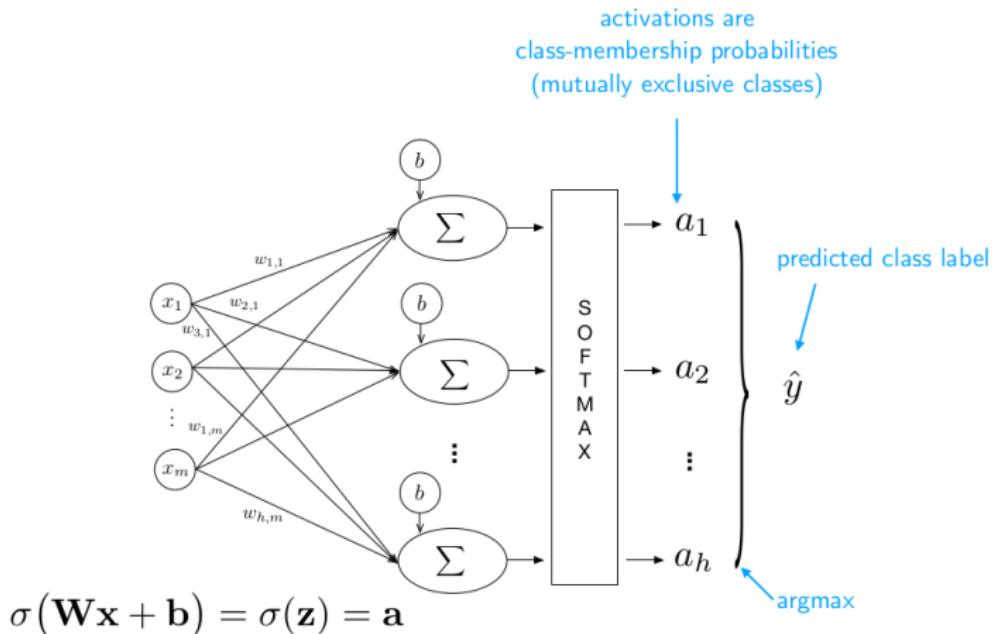
# Softmax Regression

## Multi-Class Classification with Multiple Binary Classifiers

One-vs-Rest (OvR) / One-vs-All (OvA)



# Multinomial Logistic Regression / Softmax Regression



# Softmax Activation

$$P(y = t \mid z_t^{(i)}) = \sigma_{\text{softmax}}(z_t^{(i)}) = \frac{e^{z_t^{(i)}}}{\sum_{j=1}^k e^{z_j^{(i)}}}$$

$t \in \{j \dots k\}$

$k$  is the number of class labels

(Basically, softmax is just an exponential function that normalizes the activations so that they sum up to 1)

# Loss Function

$$\mathcal{L}_{\text{binary}} = - \sum_{i=1}^n \left( y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) \right)$$

This assumes one-hot encoded labels!

$$\mathcal{L}_{\text{multiclass}} = \sum_{i=1}^n \sum_{j=1}^k -y_j^{[i]} \log \left( a_k^{[i]} \right) \quad \begin{array}{l} \text{(Multi-category) Cross Entropy} \\ \text{for } k \text{ different class labels} \end{array}$$

# Cross Entropy Loss Function Example

$$\mathbf{Y}_{\text{onehot}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{A}_{\text{softmax outputs}} = \begin{bmatrix} 0.3792 & 0.3104 & 0.3104 \\ 0.3072 & 0.4147 & 0.2780 \\ 0.4263 & 0.2248 & 0.3490 \\ 0.2668 & 0.2978 & 0.4354 \end{bmatrix}$$

(4 training examples, 3 classes)

## Cross Entropy Loss Function Example

$$\mathbf{Y}_{\text{onehot}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}_{\text{softmax outputs}} = \begin{bmatrix} 0.3792 & 0.3104 & 0.3104 \\ 0.3072 & 0.4147 & 0.2780 \\ 0.4263 & 0.2248 & 0.3490 \\ 0.2668 & 0.2978 & 0.4354 \end{bmatrix}$$

$$\begin{aligned}\mathcal{L}^{[1]} &= [(-1) \cdot \log(0.3792)] \\ &\quad + [(-0) \cdot \log(0.3104)] \\ &\quad + [(-0) \cdot \log(0.3104)] \\ &= 0.969692...\end{aligned}$$

$$\begin{aligned}\mathcal{L}^{[2]} &= [(-0) \cdot \log(0.3072)] \\ &\quad + [(-1) \cdot \log(0.4147)] \\ &\quad + [(-0) \cdot \log(0.2780)] \\ &= 0.880200...\end{aligned}$$

$$\begin{aligned}\mathcal{L}^{[3]} &= [(-0) \cdot \log(0.4263)] \\ &\quad + [(-0) \cdot \log(0.2248)] \\ &\quad + [(-1) \cdot \log(0.3490)] \\ &= 1.05268...\end{aligned}$$

$$\begin{aligned}\mathcal{L}^{[4]} &= [(-0) \cdot \log(0.2668)] \\ &\quad + [(-0) \cdot \log(0.2978)] \\ &\quad + [(-1) \cdot \log(0.4354)] \\ &= 0.831490...\end{aligned}$$

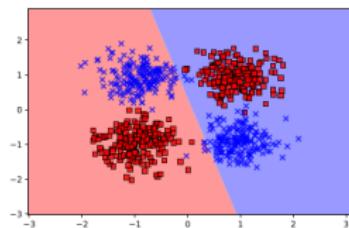
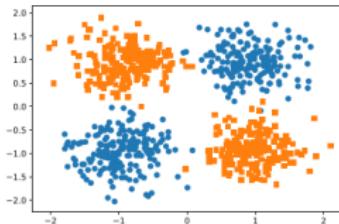
$$\mathcal{L}_{\text{multiclass}} = \frac{1}{n} \sum_{i=1}^n \sum_{j=1}^k -y_j^{[i]} \log(a_k^{[i]})$$

$\approx 0.9335$

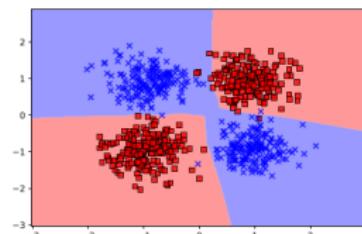


# Multilayer Perceptrons

- Problema de superfícies não lineares.



1-hidden layer MLP  
with linear activation function



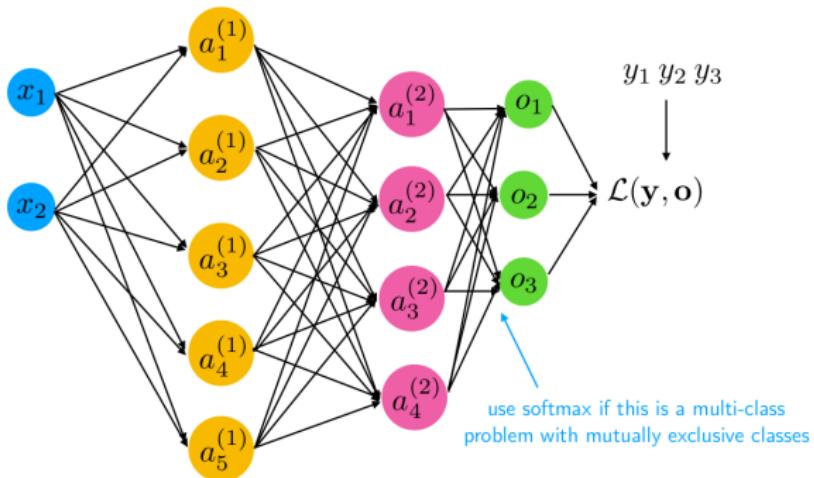
1-hidden layer MLP  
with non-linear activation function (ReLU)

# Multilayer Perceptrons

- Redes neurais feed-forward com neurônios e camadas suficientes podem aproximar qualquer função e suas derivadas.
- MLPs são aproximadores universais.

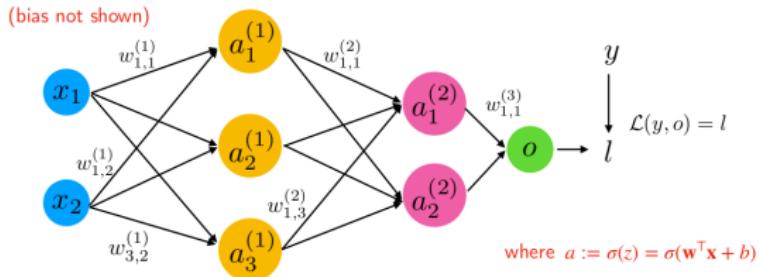
# Multilayer Perceptrons

Graph with Fully-Connected Layers  
= Multilayer Perceptron



## Graph with Fully-Connected Layers = Multilayer Perceptron

Nothing new, really



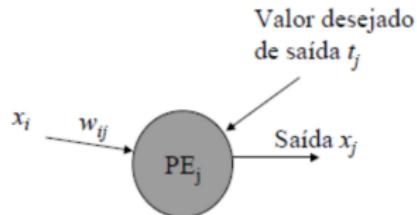
$$\begin{aligned}\frac{\partial l}{\partial w_{1,1}^{(1)}} &= \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} \quad (\text{Assume network for binary classification}) \\ &\quad + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}\end{aligned}$$

## Multi-Layer Perceptron.

- O grande desafio foi achar um algoritmo de aprendizado para atualizar dos pesos das camadas intermediarias.
- Idéia Central
  - Os erros dos elementos processadores da camada de saída (conhecidos pelo treinamento supervisionado) são retro-propagados para as camadas intermediarias

## ■ Processo de aprendizado

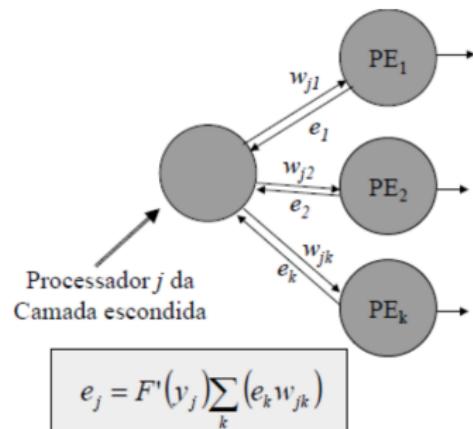
- Processador j pertence à Camada de Saída:



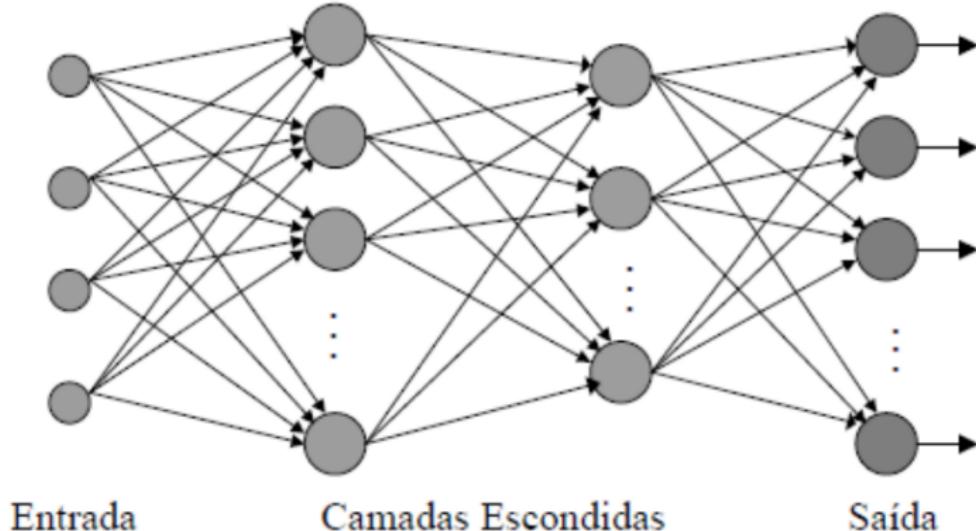
$$e_j = (t_j - x_j)F'(y_j)$$

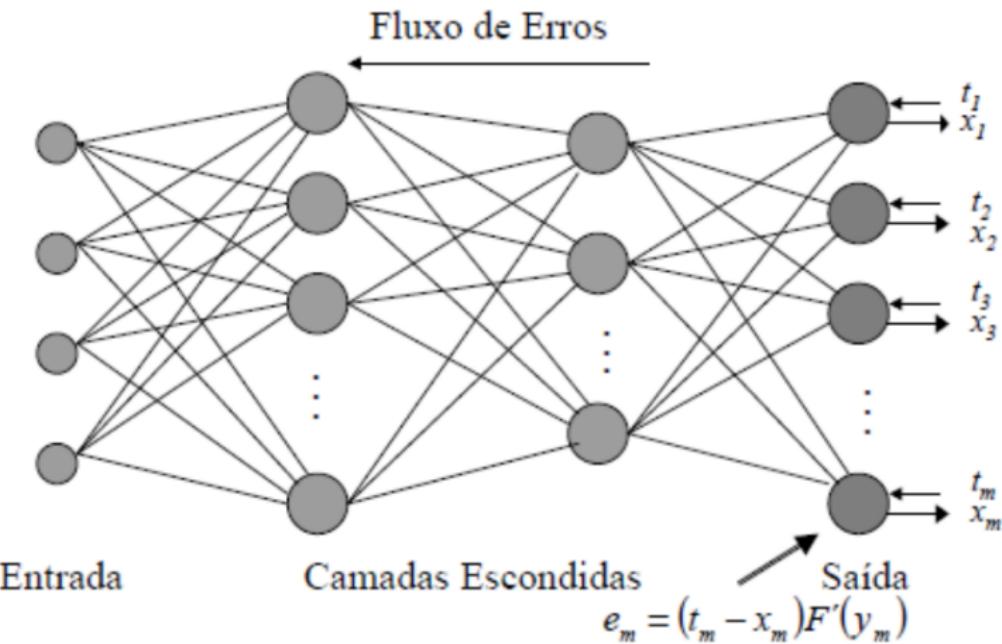
## ■ Processo de aprendizado

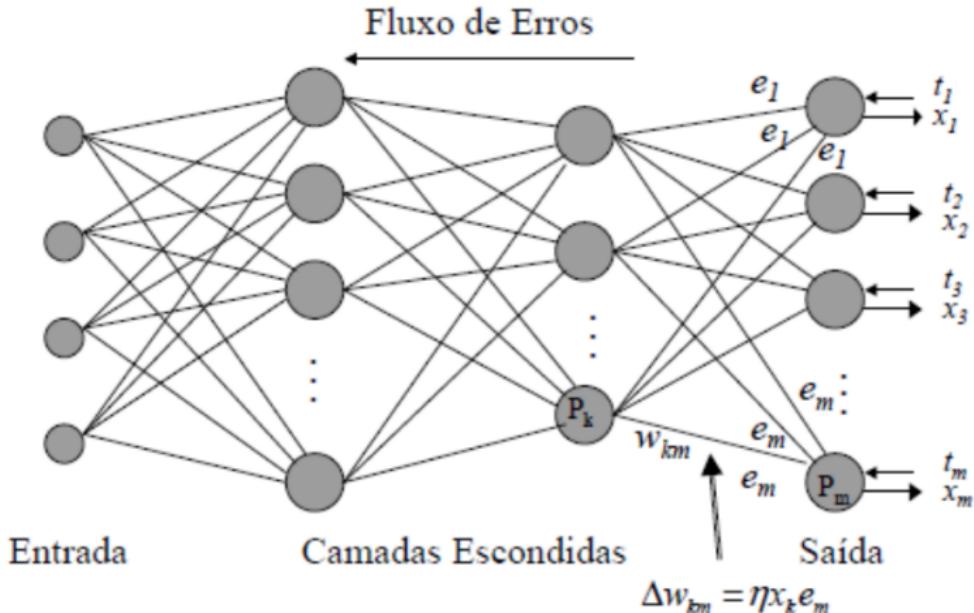
- Processador j pertence à Camada Escondida::

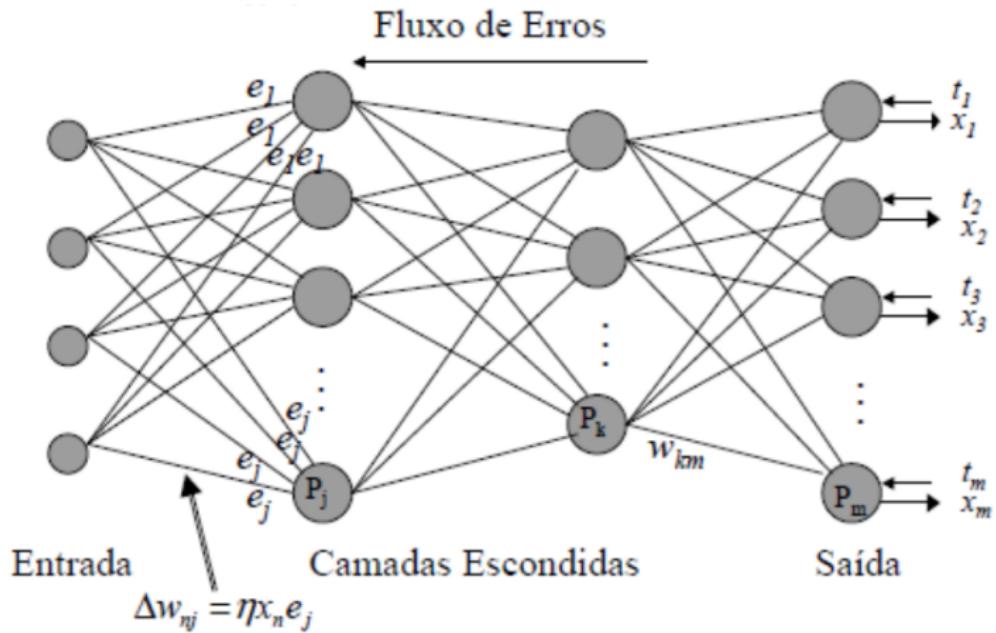


Fluxo de Dados

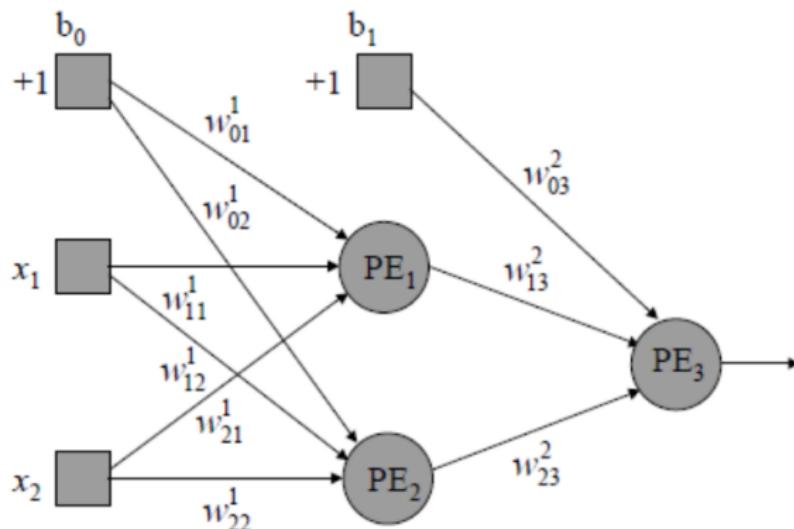








# Exemplo MLP



# Exemplo MLP

- Entrada:

- $x_1 = 1, x_2 = 0$

- Saída Desejada:

- $t_3 = 1$

- Pesos iniciais:

- $w_{ij}(0) = 0$

- Taxa de Aprendizagem:

- $\eta = 0.5$

- Função de Ativação:

$$F(y_i) = \frac{1}{1 + \exp(-y_i)}$$

- Derivada da Função de Ativação:

$$F'(y_i) = \frac{\exp(-y_i)}{[1 + \exp(-y_i)]^2}$$

# Exemplo MLP

- Algoritmo de Aprendizado:

$$w_{ij} = w_{ij} + \eta x_i e_j$$

- Camada de Saída

$$e_j = (t_j - x_j) F'(y_j)$$

- Camada Escondida

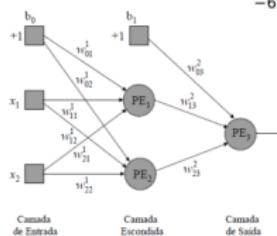
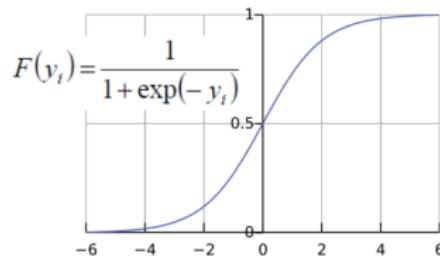
$$e_j = F'(y_j) \sum_k e_k w_{jk}$$

# Exemplo MLP

## □ Feed-Forward:

- $y_1 = 1*0 + 1*0 + 0*0 = 0$ 
  - $x_1 = F(y_1) = 0.5$
- $y_2 = 1*0 + 1*0 + 0*0 = 0$ 
  - $x_2 = F(y_2) = 0.5$
- $y_3 = 1*0 + 0.5*0 + 0.5*0 = 0$ 
  - $x_3 = F(y_3) = 0.5$

$$y_j = \sum x_i w_{ij} + \theta_j$$

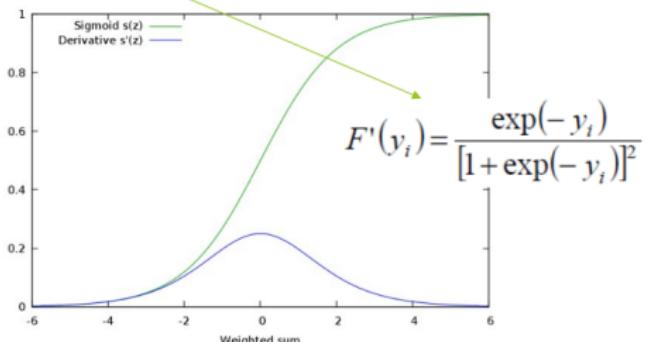
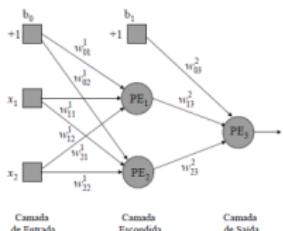


# Exemplo MLP

## □ Feed-Forward:

- $t_3 - x_3 = 1 - 0.5 = 0.5$
- $e_3 = 0.5 * 0.25 = 0.125$

$$e_j = (t_j - x_j) F'(y_j)$$

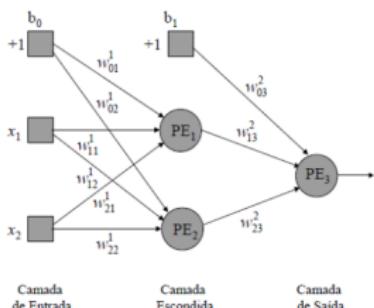


# Exemplo MLP

## Feed-Forward:

- $w^2_{03} = 0 + 0.5 * 1 * 0.125 = 0.0625$
- $w^2_{13} = 0 + 0.5 * 0.5 * 0.125 = 0.0313$
- $w^2_{23} = 0 + 0.5 * 0.5 * 0.125 = 0.0313$

$$w_{ij} = w_{ij} + \eta x_i e_j$$

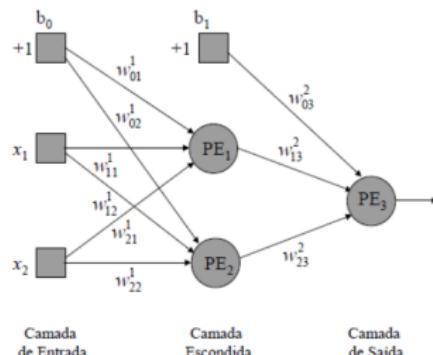


# Exemplo MLP

## □ Feed-Backward:

- $e_1 = 0.25 * (0.125 * 0.0313) = 0.00097813$
- $e_2 = 0.25 * (0.125 * 0.0313) = 0.00097813$

$$e_j = F'(y_j) \sum_k e_k w_{jk}$$

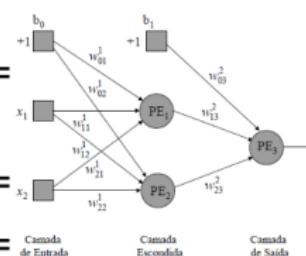


# Exemplo MLP

## □ Feed-Backward:

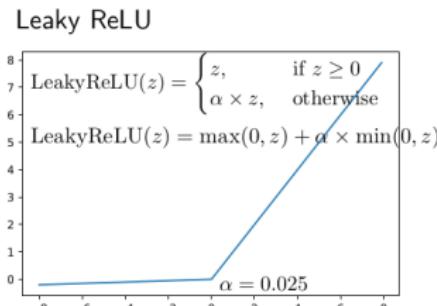
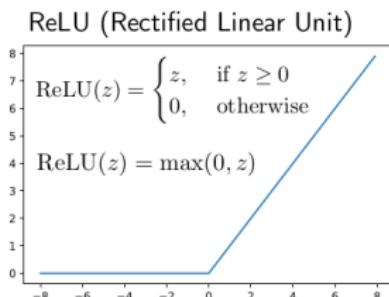
- $w^1_{01} = 0 + 0.5 * 1 * 0.00097813 = 0.00048907$
- $w^1_{02} = 0 + 0.5 * 1 * 0.00097813 = 0.00048907$
- $w^1_{11} = 0 + 0.5 * 1 * 0.00097813 = 0.00048907$
- $w^1_{12} = 0 + 0.5 * 1 * 0.00097813 = 0.00048907$
- $w^1_{21} = 0 + 0.5 * 0 * 0.00097813 = 0$
- $w^1_{22} = 0 + 0.5 * 0 * 0.00097813 = 0$

$$w_{ij} = w_{ij} + \eta x_i e_j$$



# Função de ativação Relu

- ReLU é uma abreviação para rectified linear unit, ou unidade linear retificada. Ela produz resultados no intervalo  $[0, \infty[$ .
- É a mais utilizada pelos seus bons resultados e custo computacional.



A função ReLU retorna 0 para todos os valores negativos, e o próprio valor para valores positivos. É uma função computacionalmente leve, entretanto não é centrada em zero. Como seu resultado é zero para valores negativos, ela tende a “apagar” alguns neurônios durante um passo forward, o que aumenta a velocidade do treinamento, mas por outro pode fazer com que esses neurônios “morram” e não aprendam nada se eles só receberem valores negativos. Além disso, ela pode produzir ativações explodidas, já que não possui um limite positivo. Mesmo com suas limitações, a função ReLU é hoje uma das funções de ativação mais utilizadas no treinamento de redes neurais, e não costuma ser utilizada na camada de saída.





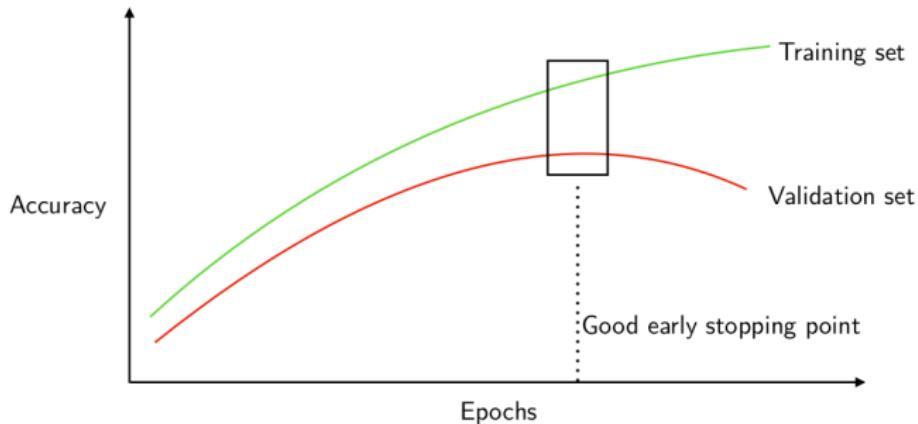
# Regularization

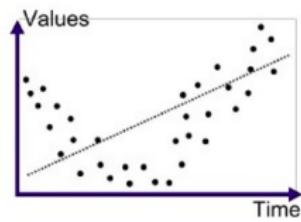
## Dataset



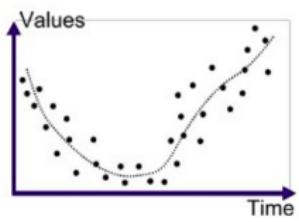
## Step 2: Early stopping (not very common anymore)

- reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point

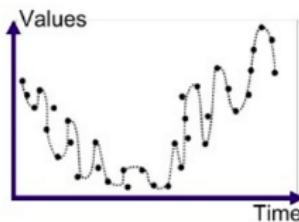




Underfitted



Good Fit/R robust



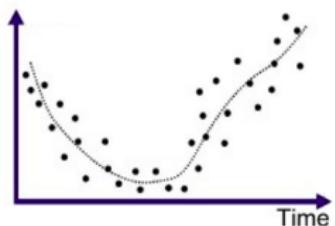
Overfitted

No contexto da aprendizagem profunda, a regularização pode ser entendido como o processo de adicionar informações/alterar a função objetivo para evitar overfitting.

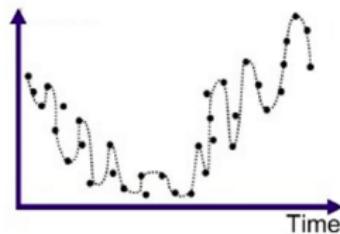
- Tipos de regularização

- $L_2$ .
- Dropout

- Regularização permite manter todas as características, mas evita que essas características tenham um efeito excessivamente grande.



$$\omega_1 x + \omega_2 x^2 + b$$



$$\omega_1 x + \omega_2 x^2 + \omega_3 x^3 + \omega_4 x^4 + b$$

- Vamos tomar  $\omega_3, \omega_4$  pequeno o suficiente ( $\approx 0$ ).
- Isso é chamado de regularização ou penalização de  $\omega_3$  e  $\omega_4$ .

- Nem sempre podemos saber quais pesos  $\omega_j$  podemos penalizar, assim em geral, regularizamos todos  $\omega_j$

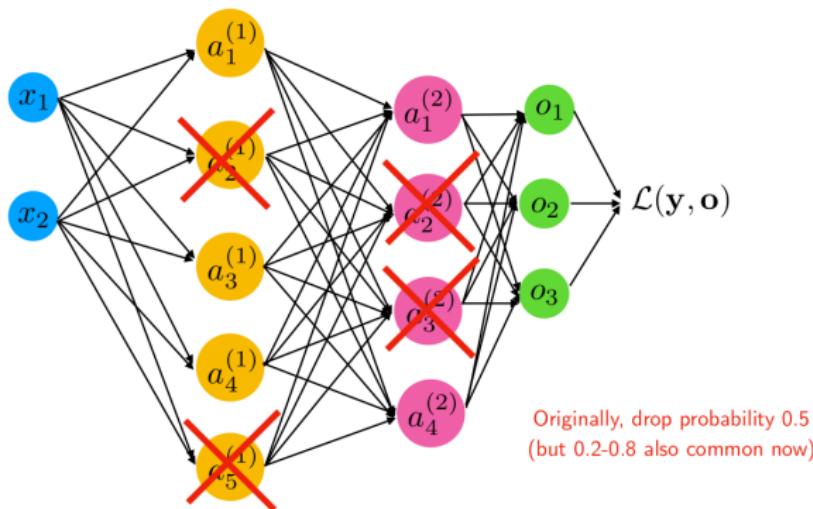
$$\min_{\omega,b} \mathcal{L}(\omega, b) = \min_{\omega,b} \left[ \frac{1}{2n} \sum_{i=1}^n \left( f_{\omega,b}(x^i) - y^i \right)^2 + \frac{\lambda}{2n} \sum_{j=1}^n \omega_j^2 \right] \quad (17)$$

onde  $\lambda$  é um hiperparâmetro regularizador.

- Se escolhermos  $\lambda \approx 0$ , não há regularização e não irá impedir o overfit.
- Se escolhermos um valor muito grande ( $\lambda \approx 10^{10}$ ), estamos colocando um peso muito alto como regularização  $(\frac{\lambda}{2n} \sum_{j=1}^n \omega_j^2)$  e a minimização desejada é escolhida com todos os pesos  $\omega_j = 0$ .

# Dropout

## ■ Dropout



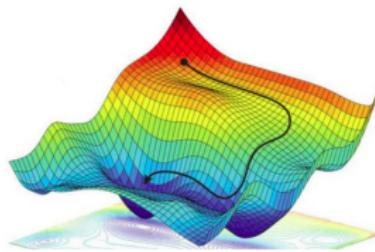
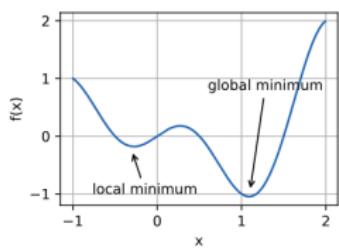
## Por que o Dropout funciona bem?

- A rede aprenderá a não depender também de conexões específicas fortemente
- Assim, considerará mais conexões (porque não pode contar com individuais)
- Os valores de peso serão mais dispersos (pode levar a menores pesos como com a norma L2)
- Nota lateral: Você certamente pode usar diferentes probabilidades de abandono em camadas diferentes (atribuindo-as proporcionais ao número de unidades em uma camada não é uma má ideia, por exemplo)

## Código: Regularization Dropout MLP Keras



# Optimização



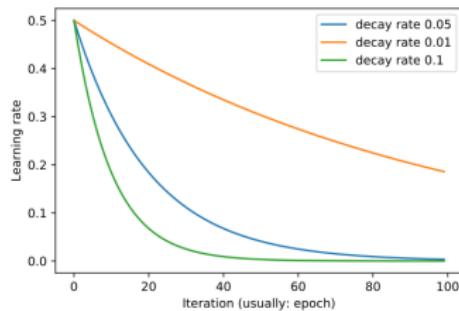
## Learning Rate Decay

Most common variants for learning rate decay:

1) Exponential Decay:

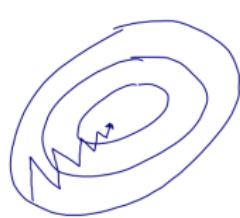
$$\eta_t := \eta_0 \cdot e^{-k \cdot t}$$

where  $k$  is the decay rate

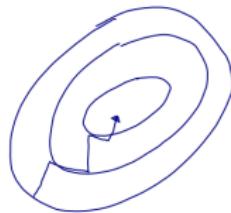


## Training with "Momentum"

- Concept: In momentum learning, we try to accelerate convergence by dampening oscillations using "velocity" (the speed of the "movement" from previous updates)



Without momentum



With momentum

## Training with "Momentum"

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

Often referred to as "velocity"  $v$

"velocity" from the previous iteration

Usually, we choose a momentum rate between 0.9 and 0.999; you can think of it as a "friction" or "dampening" parameter

Regular partial derivative/gradient multiplied by learning rate at current time step  $t$

Weight update using the velocity vector:

$$w_{i,j}(t+1) := w_{i,j}(t) - \Delta w_{i,j}(t)$$

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural Networks : The Official Journal of the International Neural Network Society*, 12(1), 145–151. [http://doi.org/10.1016/S0893-6080\(98\)00116-6](http://doi.org/10.1016/S0893-6080(98)00116-6)

## Adaptive Learning Rate via ADAM

- ADAM (Adaptive Moment Estimation) is probably the most widely used optimization algorithm in DL as of today
- It is a combination of the momentum method and RMSProp

Momentum-like term:

$$\Delta w_{i,j}(t) := \alpha \cdot \Delta w_{i,j}(t-1) + \eta \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

original momentum term

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

The diagram shows the ADAM update equations. The first equation is the momentum-like term, which is a weighted sum of the previous momentum value ( $m_{t-1}$ ) and the gradient at time  $t$ . The second equation is the actual weight update, which is the momentum term plus a learning rate ( $\eta$ ) times the gradient at time  $t$ . Red annotations highlight the 'Momentum-like term', the variable  $m_t$ , and the 'original momentum term'.

## Adaptive Learning Rate via ADAM

Momentum-like term:

$$m_t := \alpha \cdot m_{t-1} + (1 - \alpha) \cdot \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t)$$

RMSProp term:

$$r := \beta \cdot \text{MeanSquare}(w_{i,j}, t-1) + (1 - \beta) \left( \frac{\partial \mathcal{L}}{\partial w_{i,j}}(t) \right)^2$$

ADAM update:

$$w_{i,j} := w_{i,j} - \eta \frac{m_t}{\sqrt{r} + \epsilon}$$

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

codigo exemplo.

- Redes muito profundas:
  - Gradiente explosivo ou nulo.
  - Pré treinar uma camada por vez, após fazer o ajuste fino em toda as camadas.
- Exemplo código de DL.

codigo: opt mlp



# Solving differential equations using neural networks

- Feed-forward neural networks with enough neurons and layers can approximate any function and its derivatives.
- MLP are universal approximators.

# Solving an ODE using neural networks

I'm very new to deep learning (coming from a math PDE background), but I'm trying to solve some ODEs using a neural network (via tensorflow). I've solved some simple ones like  $u'(x) + u(x) = f(x)$  with no problem, but I'm trying something a bit harder now:

$$u''(x) - xu(x) = 0$$

with the initial conditions  $u(0) = A$  and  $u'(0) = B$

I'm mostly following this paper, and my solution is written as

$$u_N(x, \Theta) = A + Bx + x^2 N(x, \Theta),$$

where  $N(x, \omega)$  is the output of the neural net. The loss function I'm using is just the residual of the ODE in a mean square sense, so it's pretty crude:

$$\mathcal{L}(\Theta) = \frac{1}{N} \sum_{j=1}^N (u_N''(x, \Theta) - xu_N(x, \Theta))^2.$$

I'm having a lot of trouble getting a good numerical solution to this particular equation. You can see a typical result below (orange is the exact solution, blue is my solution).

Solving the problem with Stochastic Gradient Descent:

$$\Theta_k = \Theta_{k+1} + \alpha_k \sum_{i=1}^N \nabla_{\Theta} \mathcal{L}(\Theta) \quad (18)$$

codigo: ode mlp

We describe the PINN approach for approximating the solution  $u : [0, T] \times \mathcal{D} \rightarrow \mathbb{R}$  of an evolution equation

$$\partial_t u(t, x) + \mathcal{N}[u](t, x) = 0, \quad (t, x) \in (0, T] \times \mathcal{D}, \quad (19)$$

$$u(0, x) = u_0(x) \quad x \in \mathcal{D}, \quad (20)$$

where  $\mathcal{N}$  is a nonlinear differential operator acting on  $u$ ,  $\mathcal{D} \subset \mathbb{R}^d$  a bounded domain,  $T$  denotes the final time and  $u_0 : \mathcal{D} \rightarrow \mathbb{R}$  the prescribed initial data. Although the methodology allows for different types of boundary conditions, we restrict our discussion to the inhomogeneous Dirichlet case and prescribe

$$u(t, x) = u_b(t, x) \quad (t, x) \in (0, T] \times \partial\mathcal{D}, \quad (21)$$

where  $\partial\mathcal{D}$  denotes the boundary of the domain  $\mathcal{D}$  and  $u_b : (0, T] \times \partial\mathcal{D} \rightarrow \mathbb{R}$  the given boundary data.

The method constructs a neural network approximation

$$u_\theta(t, x) \approx u(t, x) \quad (22)$$

of the solution of nonlinear PDE, where  $u_\theta : [0, T] \times \mathcal{D} \rightarrow \mathbb{R}$  denotes a function realized by a neural network with parameters  $\theta$ .

The continuous time approach for the parabolic PDE is based on the (strong) residual of a given neural network approximation  $u_\theta : [0, T] \times \mathcal{D} \rightarrow \mathbb{R}$  of the solution  $u$ , i.e.,

$$r_\theta(t, x) := \partial_t u_\theta(t, x) + \mathcal{N}[u_\theta](t, x). \quad (23)$$

To incorporate this PDE residual  $r_\theta$  into a loss function to be minimized, PINNs require a further differentiation to evaluate the differential operators  $\partial_t u_\theta$  and  $\mathcal{N}[u_\theta]$ . Thus the PINN term  $r_\theta$  shares the same parameters as the original network  $u_\theta(t, x)$ , but respects the underlying "physics" of the nonlinear PDE. Both types of derivatives can be easily determined through automatic differentiation with current state-of-the-art machine learning libraries, e.g., TensorFlow or PyTorch.

The PINN approach for the solution of the initial and boundary value problem now proceeds by minimization of the loss functional

$$\phi_\theta(X) := \phi_\theta^r(X^r) + \phi_\theta^0(X^0) + \phi_\theta^b(X^b), \quad (24)$$

where  $X$  denotes the collection of training data and the loss function  $\phi_\theta$  contains the following terms:

- the mean squared residual

$$\phi_\theta^r(X^r) := \frac{1}{N_r} \sum_{i=1}^{N_r} |r_\theta(t_i^r, x_i^r)|^2 = \frac{1}{N_r} \sum_{i=1}^{N_r} |\partial_t u_\theta(t, x) + \mathcal{N}[u_\theta](t, x)|^2$$

in a number of collocation points

$X^r := \{(t_i^r, x_i^r)\}_{i=1}^{N_r} \subset (0, T] \times \mathcal{D}$ , where  $r_\theta$  is the physics-informed neural network.

- the mean squared misfit with respect to the initial and boundary conditions

$$\phi_\theta^0(X^0) := \frac{1}{N_0} \sum_{i=1}^{N_0} \left| u_\theta \left( t_i^0, x_i^0 \right) - u_0 \left( x_i^0 \right) \right|^2 \quad \text{and}$$

$$\phi_\theta^b(X^b) := \frac{1}{N_b} \sum_{i=1}^{N_b} \left| u_\theta \left( t_i^b, x_i^b \right) - u_b \left( t_i^b, x_i^b \right) \right|^2$$

in a number of points  $X^0 := \{(t_i^0, x_i^0)\}_{i=1}^{N_0} \subset \{0\} \times \mathcal{D}$  and  $X^b := \{(t_i^b, x_i^b)\}_{i=1}^{N_b} \subset (0, T] \times \partial\mathcal{D}$ , where  $u_\theta$  is the neural network approximation of the solution  $u: [0, T] \times \mathcal{D} \rightarrow \mathbb{R}$ .

Note that the training data  $X$  consists entirely of time-space coordinates.

Solving the problem with Stochastic Gradient Descent:

$$\Theta_k = \Theta_{k+1} + \alpha_k \sum_{i=1}^N \nabla_{\Theta} \phi(X, \Theta) \quad (25)$$

---

**Procedure 2.1** The PINN algorithm for solving differential equations.

---

- Step 1 Construct a neural network  $\hat{u}(\mathbf{x}; \boldsymbol{\theta})$  with parameters  $\boldsymbol{\theta}$ .
  - Step 2 Specify the two training sets  $\mathcal{T}_f$  and  $\mathcal{T}_b$  for the equation and boundary/initial conditions.
  - Step 3 Specify a loss function by summing the weighted  $L^2$  norm of both the PDE equation and boundary condition residuals.
  - Step 4 Train the neural network to find the best parameters  $\boldsymbol{\theta}^*$  by minimizing the loss function  $\mathcal{L}(\boldsymbol{\theta}; \mathcal{T})$ .
-

Código: PDE mlp

Comparison between PNNs(physics-informed neural networks) and FEM.(finite element method).

- In FEM we approximate the solution  $u$  by a piecewise polynomial with point values to be determined, while in PINNs we construct a neural network as the surrogate model parameterized by weights and biases.
- FEM typically requires a mesh generation, while PINN is totally mesh-free, and we can use either a grid or random points.
- FEM converts a PDE to an algebraic system, using the stiffness matrix and mass matrix, while PINN embeds the PDE and boundary conditions into the loss function.
- In the last step, the algebraic system in FEM is solved exactly by a linear solver, but the network in PINN is learned by a gradient-based optimizer.

At a more fundamental level, PINNs provide a nonlinear approximation to the function and its derivatives whereas FEM represent a linear approximation.

## Problema em aberto!

- $\mathcal{F} \Rightarrow$  Família de todas as funções que pode ser representada pela nossa rede.
- $u$  é nossa solução, mas é improvável que ela pertença a  $\mathcal{F}$ .
- Definimos então  $u_{\mathcal{F}} = \arg \min_{f \in \mathcal{F}} \|f - u\|$ , onde  $u_{\mathcal{F}}$  é a melhor solução que se aproxima de  $u$ .
- Como a rede neural é treinada apenas no conjunto de treinamento  $T$ , definimos  $u_T = \arg \min_{f \in \mathcal{F}} \mathcal{L}(f, T)$ , onde a função de custo está no mínimo global.

- Obter  $u_T$  é sempre intratável e nossos otimizadores nos retorna  $\hat{u}_T$
- Erro Total  $\varepsilon$ :

$$\varepsilon = \|\hat{u}_T - u\| \leq \|\hat{u}_T - u_T\| + \|u_T - u_{\mathcal{F}}\| + \|u_{\mathcal{F}} - u\|$$

$$\varepsilon = \varepsilon_{opt} + \varepsilon_{gen} + \varepsilon_{app}$$

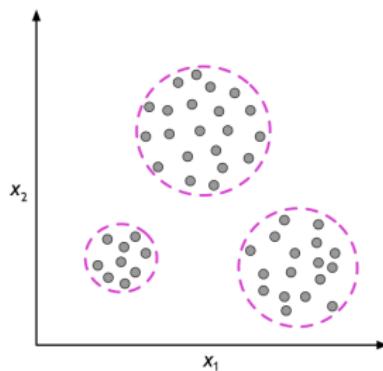
- $\varepsilon_{app} \Rightarrow u_{\mathcal{F}} \longrightarrow u$
- $\varepsilon_{gen}$  Mede a capacidade de generalização da rede, quando o erro de generalização domína, temos o "overfitting".
- $\varepsilon_{opt}$  Vem da complexidade da função de custo e da configuração de otimização ( Taxa de apredizado e números de interação)
- Quantificar esses 3 erros ainda é um problema em aberto.



# Unsupervised Learning

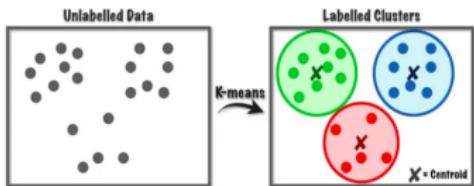
## Unsupervised Learning 2: Clustering

Assigning group memberships to unlabelled examples (instances, data points)

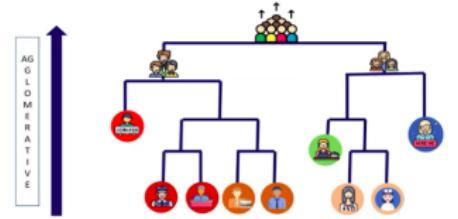


**Source:** Raschka and Mirjaliy (2019). *Python Machine Learning, 3rd Edition*

- Clustering.
  - K mean
  - Hierarchical



K mean



Agglomerative Clustering Example (Image taken from YouTube video on Hierarchical clustering by Edureka.)

# k-means

It is used in the grouping task.

Given a set  $(x_1, x_2, \dots, x_n)$ , where each observation is a real value of the D-dimension. k-means tends to group  $n$  points in ( $k \leq n$ ) together  $S = (S_1, S_2, \dots, S_K)$  that minimizes the quadratic sum within the group:

$$\arg \min_S \sum_{i=1}^k \|S_i|VarS_i\| = \arg \min_S \sum_{i=1}^k \|\boldsymbol{x}_i - \boldsymbol{\mu}_i\|^2 \quad (26)$$

where  $\boldsymbol{\mu}_i$  is the mean (also called centroid) of points in  $S_i$ , i.e.

$$\boldsymbol{\mu}_i = \frac{1}{|S_i|} \sum_{x \in S_i}^k \boldsymbol{x} \quad (27)$$

- First we define the grouping number.
- Then we define a random centroid.
- Calculate the shortest distance centroid for each point.  
Each point will belong to the closest centroid.
- Now we must reposition the centriode, the new position of the centroid must be the average of the position of all points in the cluster.

[https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/K-means\\_convergence.gif/617px-K-means\\_convergence.gif](https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/K-means_convergence.gif/617px-K-means_convergence.gif)

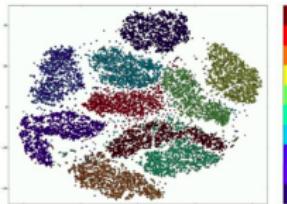
- Dimensionality reduction.
  - PCA
  - t-SNE
  - Umap

Dados de alta dimensionalidade

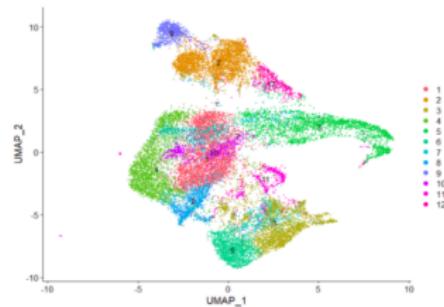
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9
```

t-SNE

Mapeamento em duas dimensões



t-SNE



Umap

- t-Stochastic Neighbour Embedding (t-SNE):

The idea of t-SNE is to map data points in the original high-dimensional space ( $\mathcal{X} = \{x_1, x_2, \dots, x_N\}$ ) to points in a low-dimensional space ( $\mathcal{Y} = \{y_1, y_2, \dots, y_N\}$ ), while keeping the similarity among the points.

$$\begin{aligned} p_{j|i} &= \frac{e^{-\|x_i - x_j\|^2 / 2\sigma_i^2}}{\sum_{k \neq l} e^{-\|x_k - x_l\|^2 / 2\sigma_i^2}}, \\ p_{i|i} &= 0 \end{aligned} \tag{28}$$

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N} \tag{29}$$

$$\begin{aligned} q_{j|i} &= \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_k \sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}, \\ q_{i|i} &= 0 \end{aligned} \quad (30)$$

$$C(\{y_i\}) = \sum_i KL(P_i \| Q_i) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}, \quad (31)$$

where  $P_i$  represents the probability distribution over all points given by  $x_i$  and  $Q_i$  represents the probability distribution over all points given by  $y_i$ .

The gradient of the cost function for  $y_i$  described

$$\begin{aligned} \frac{\partial C}{\partial y_i} &= 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \\ &\quad \| y_k - y_l \| ^2)^{-1}), \end{aligned} \quad (32)$$

where  $p \approx q$ , we have  $\frac{\partial C}{\partial y_i} \approx 0$ .

The gradient is initialised by randomly sampling the mapped points. In order to speed up the optimisation and avoid poor local minima on the error surface, a relatively large momentum term is added to the gradient,

$$\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\partial C}{\partial y_i} + a(t)(\mathcal{Y}(t-1) + \mathcal{Y}^{(t-2)}), \quad (33)$$

where  $\mathcal{Y}^{(t)}$  indicates iteration solution  $t$ ,  $\eta$  is the learning rate, and  $a(t)$  represents the moment in  $t$ .

[https:](https://leowyy.github.io/public/graphsne/graphsne.gif)

//leowyy.github.io/public/graphsne/graphsne.gif

Exemplo código: tsne

## **Temporary page!**

$\text{\LaTeX}$  was unable to guess the total number of pages correctly.  
As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because  $\text{\LaTeX}$  now knows how many pages to expect for this document.