Sidney Tiago da Silva

Física/UFPR

January 23, 2024

Contents

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs

10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

Bibliotecas

NumPy

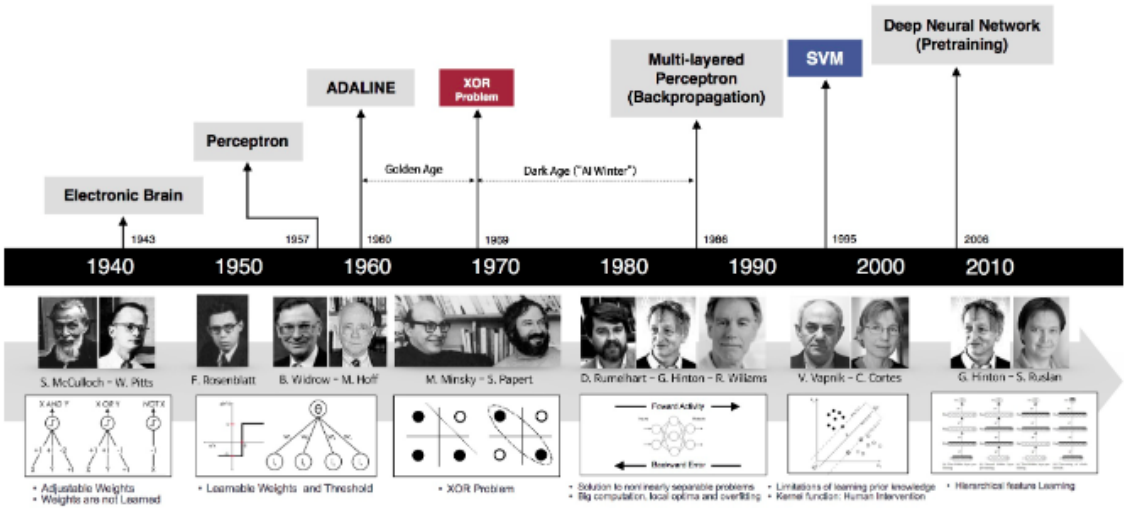
1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

Introduction

What Is Machine Learning?

A short overview before we jump into Deep Learning

History


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Williams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to non linearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

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"

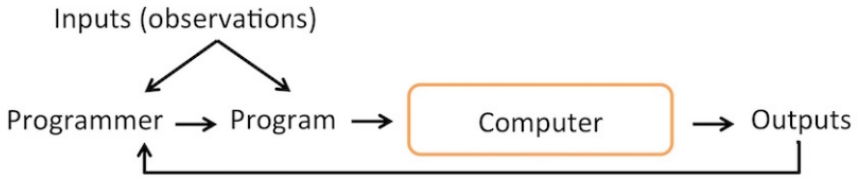
— Arthur L. Samuel, AI pioneer, 1959

[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.")

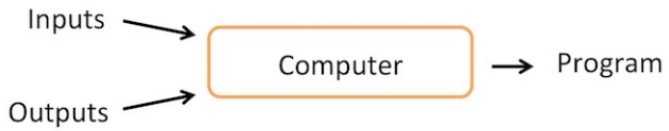
Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

The Traditional Programming Paradigm

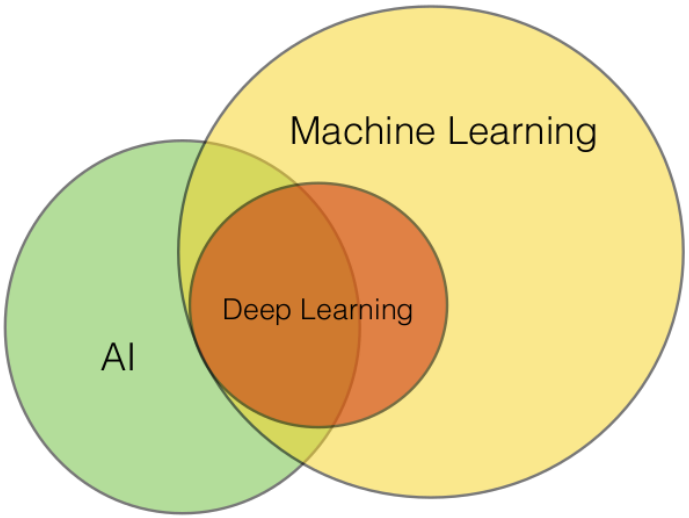


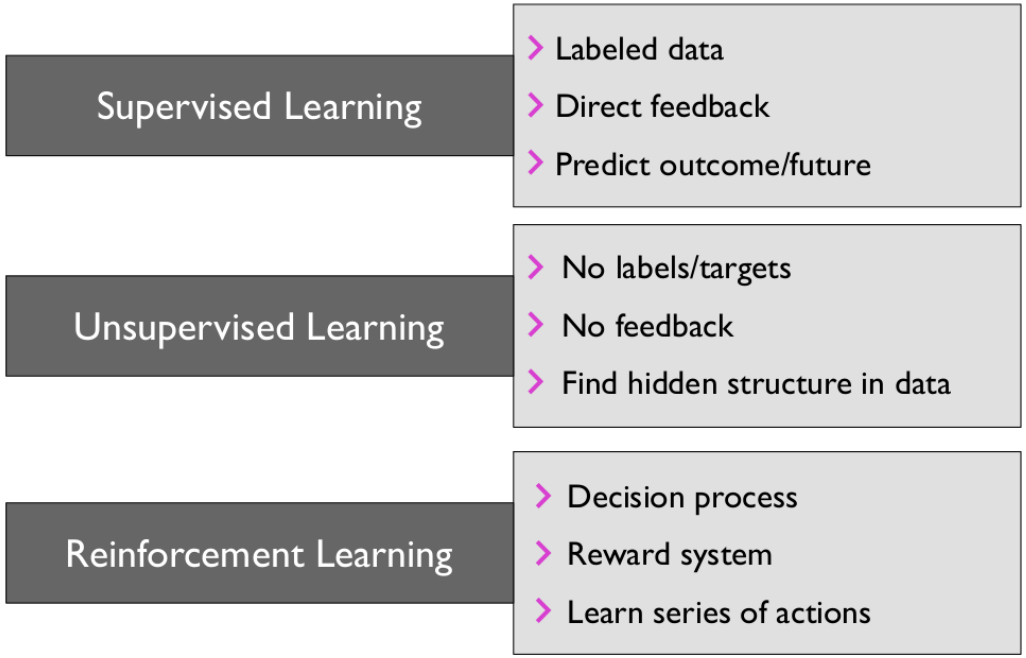
Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed
– Arthur Samuel (1959)

Machine Learning



The Connection Between Fields





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

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

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.

$$h : \mathcal{X} \longrightarrow \mathcal{Y}, \tag{1}$$

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

$$\mathbb{D} : \mathbb{R}^m \longrightarrow R \tag{2}$$

given the training set

$$\mathbb{D} : \langle x^{[i]}, y^{[i]} \rangle, i = 1, ..., n, \tag{3}$$

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

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

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

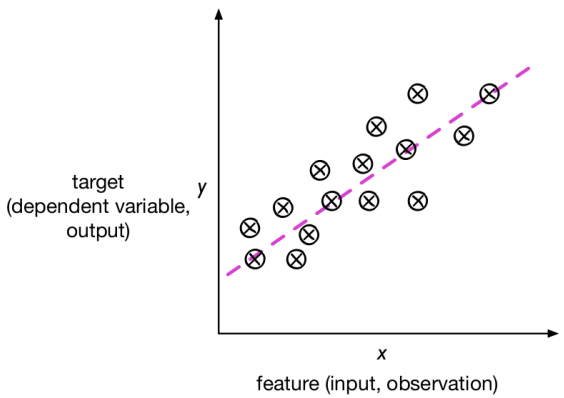
where stores n training examples and m the feature number.

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 *ith* training example and a regular underline notation to refer to the jth characteristic. The corresponding labels are represented in a column vector $\mathbf{y}$, $\mathbf{y} \in \mathbb{R}^m$

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

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

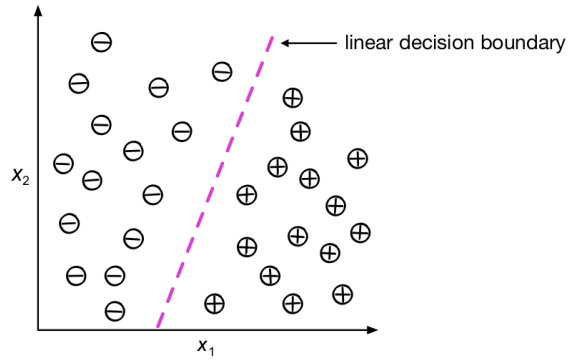
Supervised Learning 1: Regression



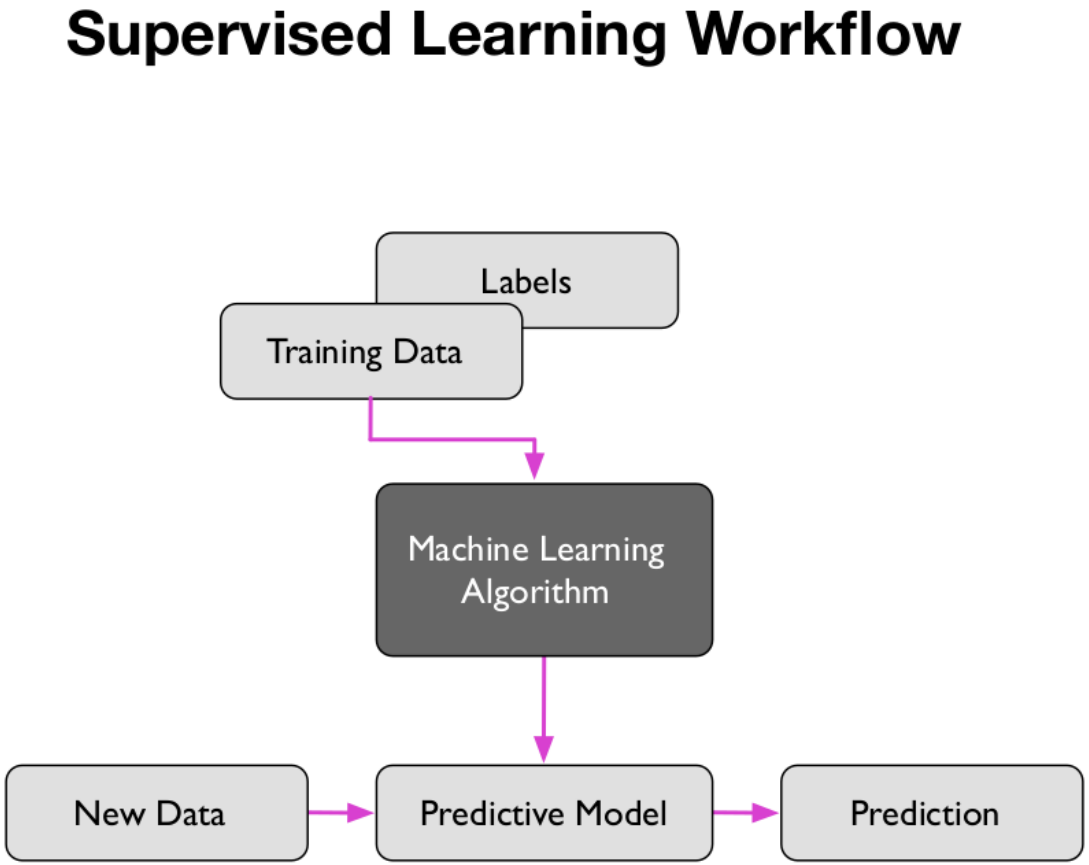
Supervised Learning 2: Classification

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

What are the class labels (y's)?



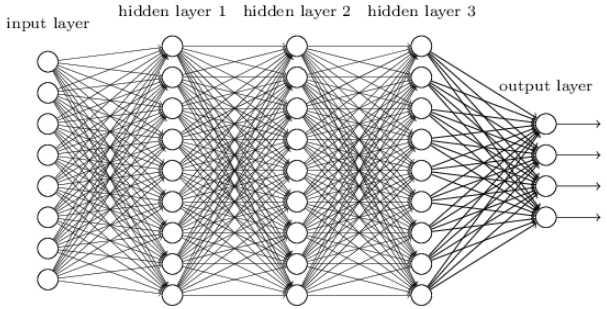
Training

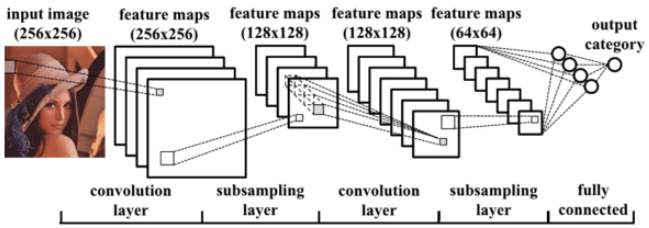

Supervised Learning Workflow
Labels
Training Data
Machine Learning Algorithm
New Data
Predictive Model
Prediction

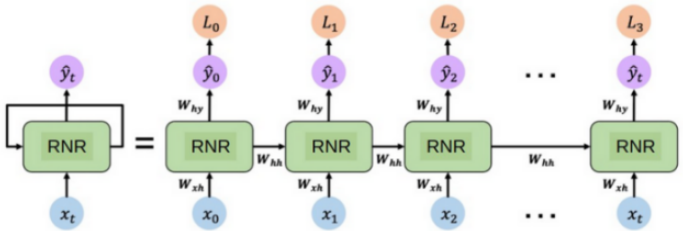
1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

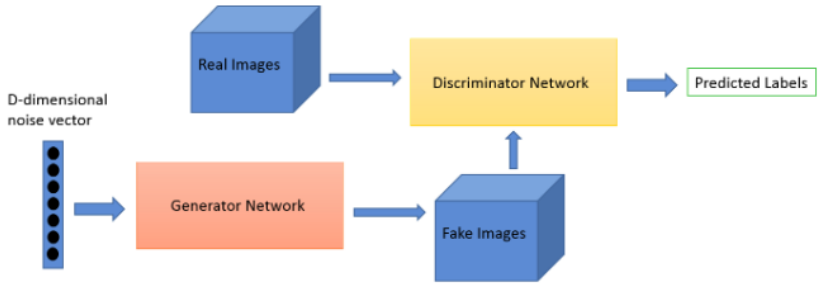
Deep Learning

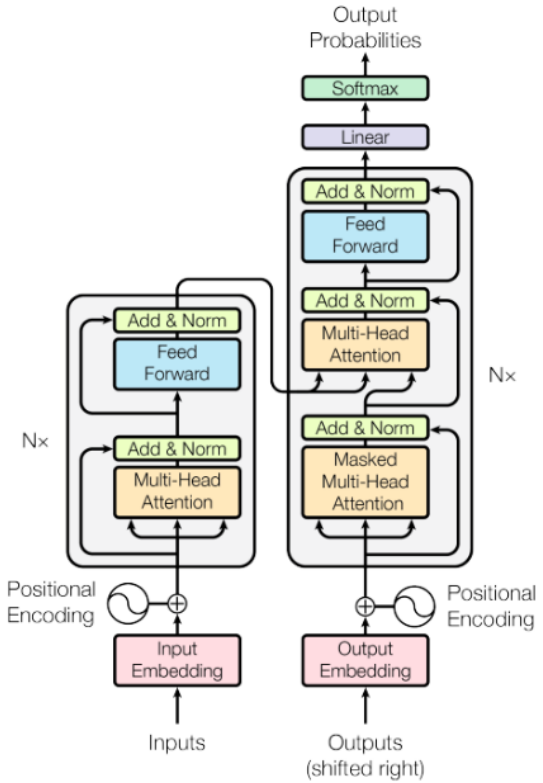
- Multilayer perceptron
- Convolutional networks
  - VGG16
  - ResNet
  - Inception
- Recurring networks
  - LSTM
  - GRU
  - Echo States
- Advanced architectures
  - Transformers
  - GANs – Generative Adversarial Networks


input layer
hidden layer 1
hidden layer 2
hidden layer 3
output layer


input image (256x256)
feature maps (256x256)
feature maps (128x128)
feature maps (128x128)
feature maps (64x64)
output category
convolution layer
subsampling layer
convolution layer
subsampling layer
fully connected

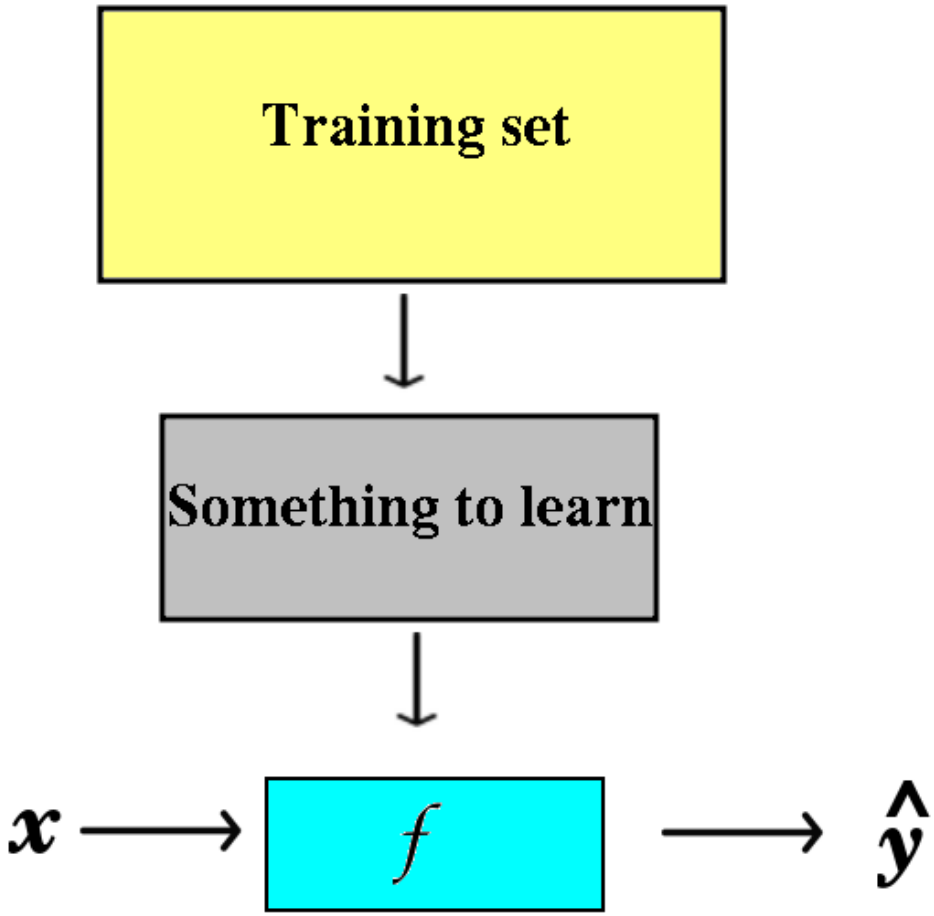

RNR
RNR
RNR
RNR
RNR


Real Images
Discriminator Network
Predicted Labels
D-dimensional
noise vector
Generator Network
Fake Images

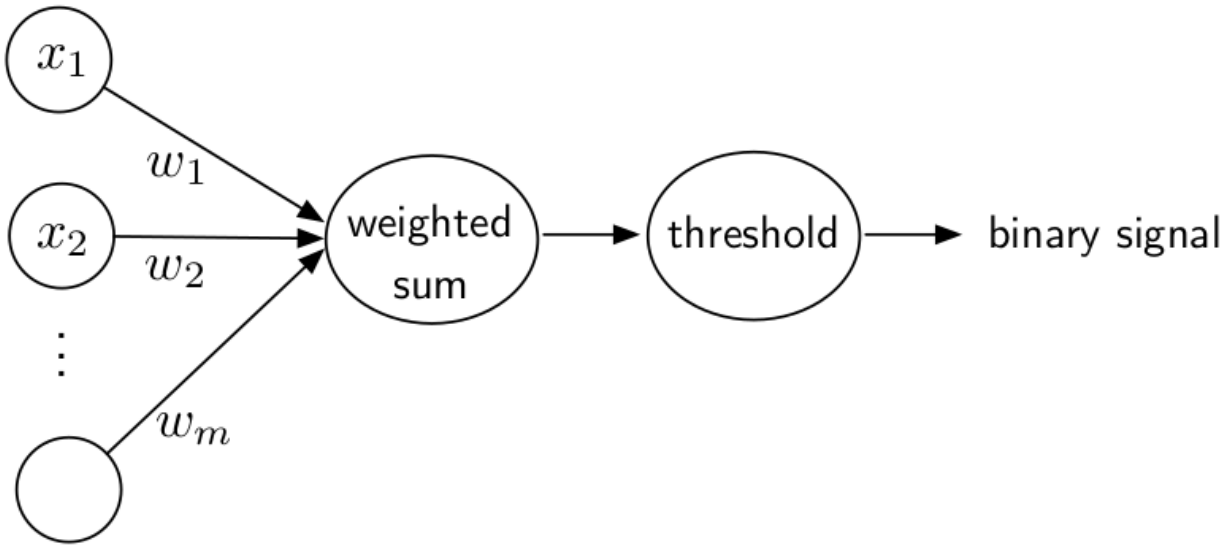

Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

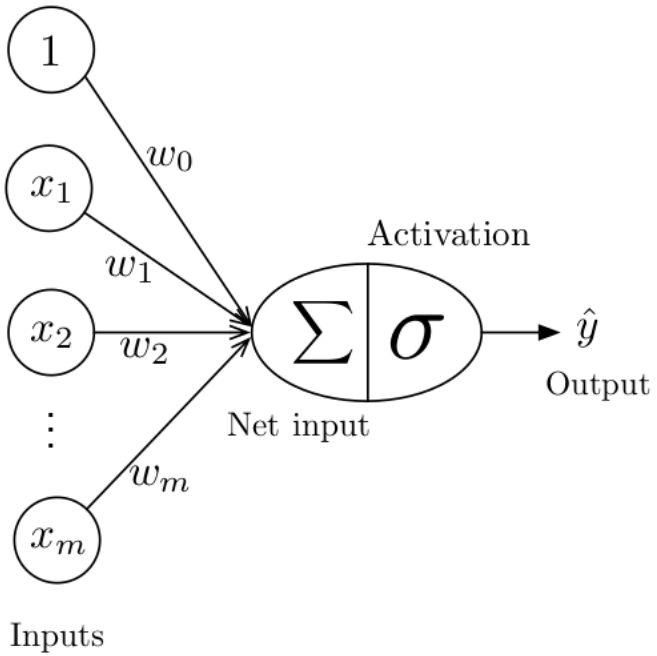
Perceptron



Perceptron



General Notation for Single-Layer Neural Networks



Vector dot product

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

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

$$w_0 = -\theta$$

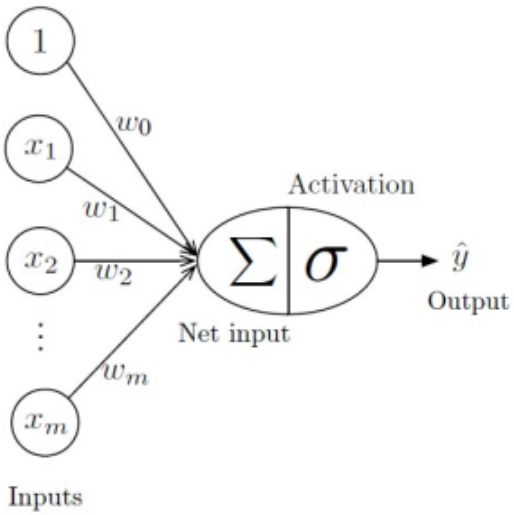
The Perceptron Learning Algorithm

Let

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

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

Simulação do Operador Lógico AND



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

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

$$w_0 = -\theta$$

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) $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]T}\mathbf{w}\right)$
      (b) $\text{err} := (y^{[i]} - \hat{y}^{[i]})$
      (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 $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]T}\mathbf{w})$ $\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$
  - z = 1*0+0*0*+0*0 = 0
  - $\hat{y} = \sigma(0) = 0$
- Calcular o erro $\text{err} := (y^{[i]} - \hat{y}^{[i]})$
  - err = 0-0 = 0
- Atualizar os pesos $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}$
  - $w_0 = 0+0*1 = 0$
  - $w_1 = 0+0*0 = 0$
  - $w_2 = 0+0*0 = 0$

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

1 epoch

- Entrada 1
  - ŷ = σ(1*0+0*0+0*0) = σ(0) = 0
  - Err = 0-0 = 0
- Entrada 2
  - ŷ = σ(1*0+0*0+1*0) = σ(0) = 0
  - Err = 0-0 = 0
- Entrada 3
  - ŷ = σ(1*0+1*0+0*0) = σ(0) = 0
  - Err = 0-0 = 0
- Entrada 4
  - ŷ = σ(1*0+1*0+1*0) = σ(0) = 0
  - Err = 1-0 = 1
  - w_0 = 0+1*1 = 1
  - w_1 = 0+1*1 = 1
  - w_2 = 0+1*1 = 1

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

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

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

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

2 epoch

- Entrada 1
  - $\hat{y}$ = σ(1*1+0*1+0*1) = σ(1) = 1
  - Err = 0-1 = -1
  - w_0 = 1+(-1)*1 = 0
  - w_1 = 1+(-1)*0 = 1
  - w_2 = 1+(-1)*0 = 1
- Entrada 2
  - $\hat{y}$ = σ(1*0+0*1+1*1) = σ(1) = 1
  - Err = 0-1 = -1
  - w_0 = 0+(-1)*1 = -1
  - w_1 = 1+(-1)*0 = 1
  - w_2 = 1+(-1)*1 = 0
- Entrada 3
  - $\hat{y}$ = σ(1*(-1)+1*1+0*0) = σ(0) = 0
  - Err = 0-0 = 0
- Entrada 4
  - $\hat{y}$ = σ(1*(-1)+1*1+1*0) = σ(0) = 0
  - Err = 1-0 = 1
  - w_0 = (-1)+1*1 = 0
  - w_1 = 1+1*1 = 2
  - w_2 = 0+1*1 = 1

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

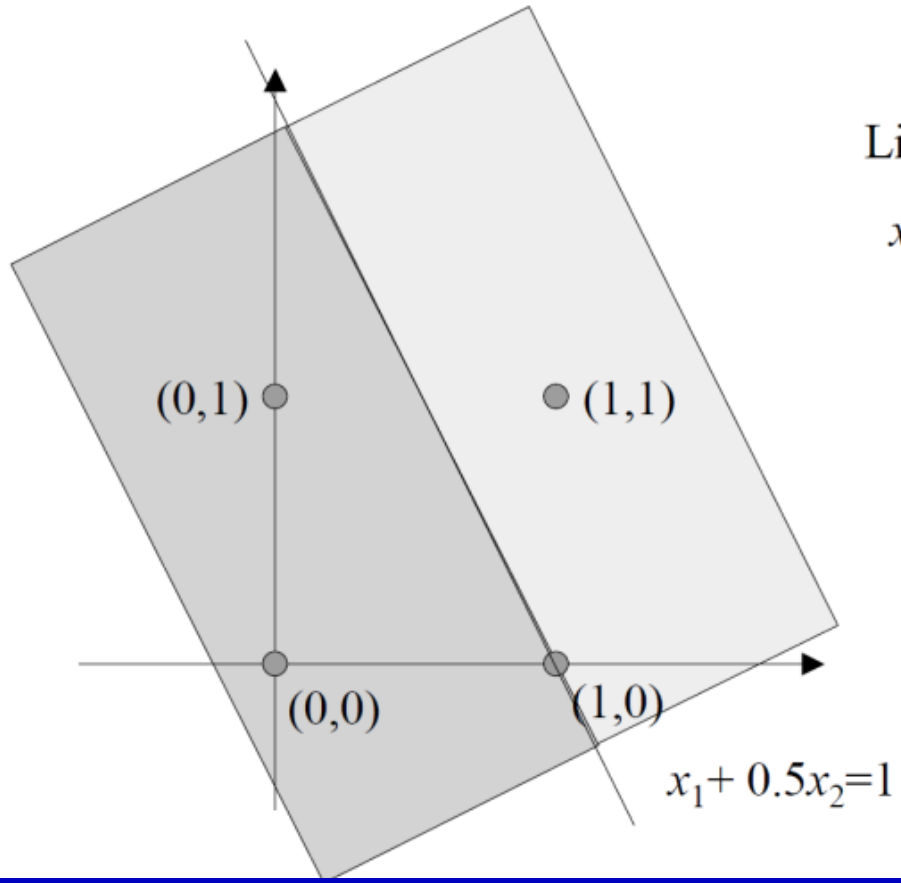
$$\sigma(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

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

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

Após 5 épocas $w_0 = -2,\ w_1 = 2,\ w_2 = 1$

$w_0 = -1,\ w_1 = 1,\ w_2 = 0{,}5$

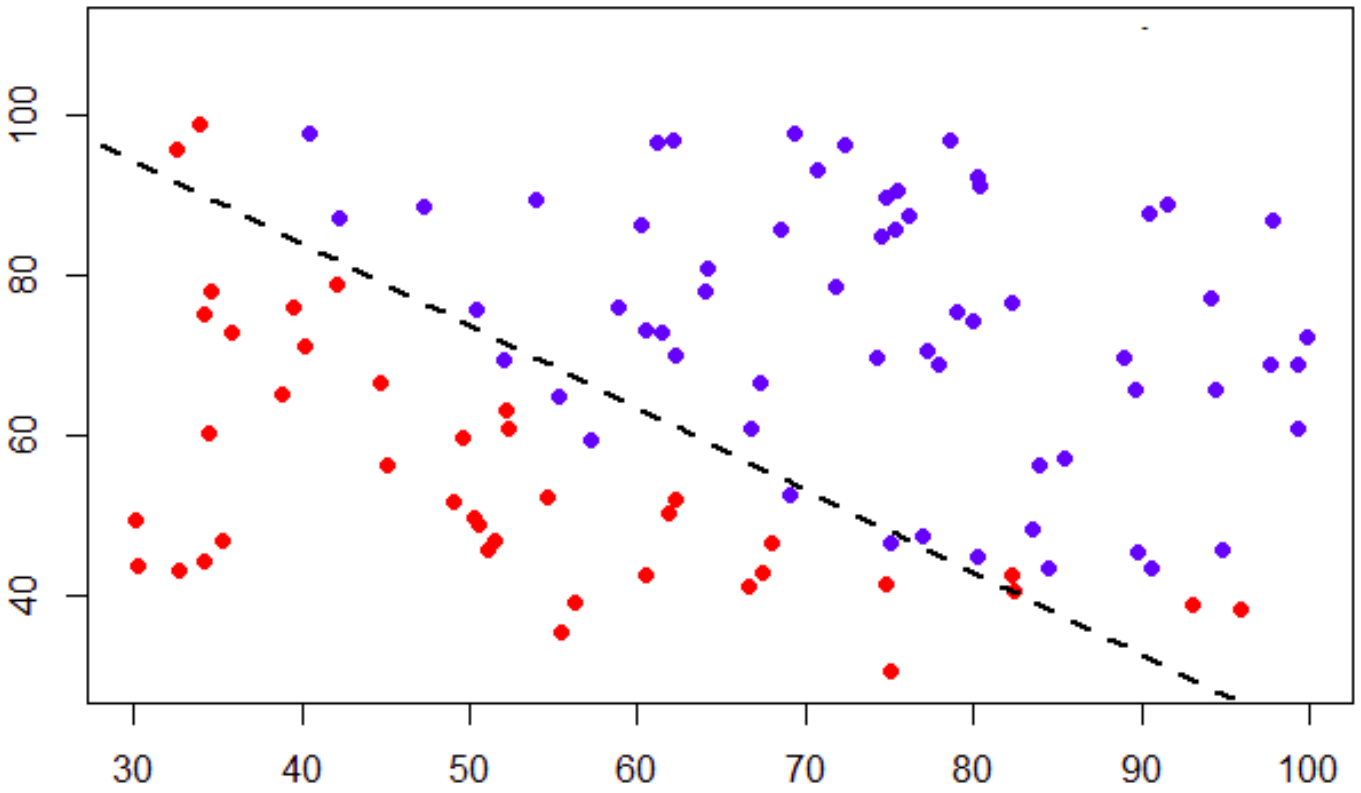


Linha de Decisão:

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

$$x_1 + 0.5\, x_2 = 1$$

- Perceptron
  - Superfície de decisão linear



- Model

$$\hat{y} = f(\boldsymbol{x}, \Theta) \tag{5}$$

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

Função de Perda

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

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

  - 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.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n \tag{7}$$

$\hat{y}$ => Valor previsto.
β => Pesos.
x => Entradas.

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

Erro quadrático Médio

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

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

$$\mathcal{L}(\Theta) = \frac{1}{2n}\sum_{i}^{n}(f(\boldsymbol{x}^i, \Theta) - y^i)^2 \tag{9}$$

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

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

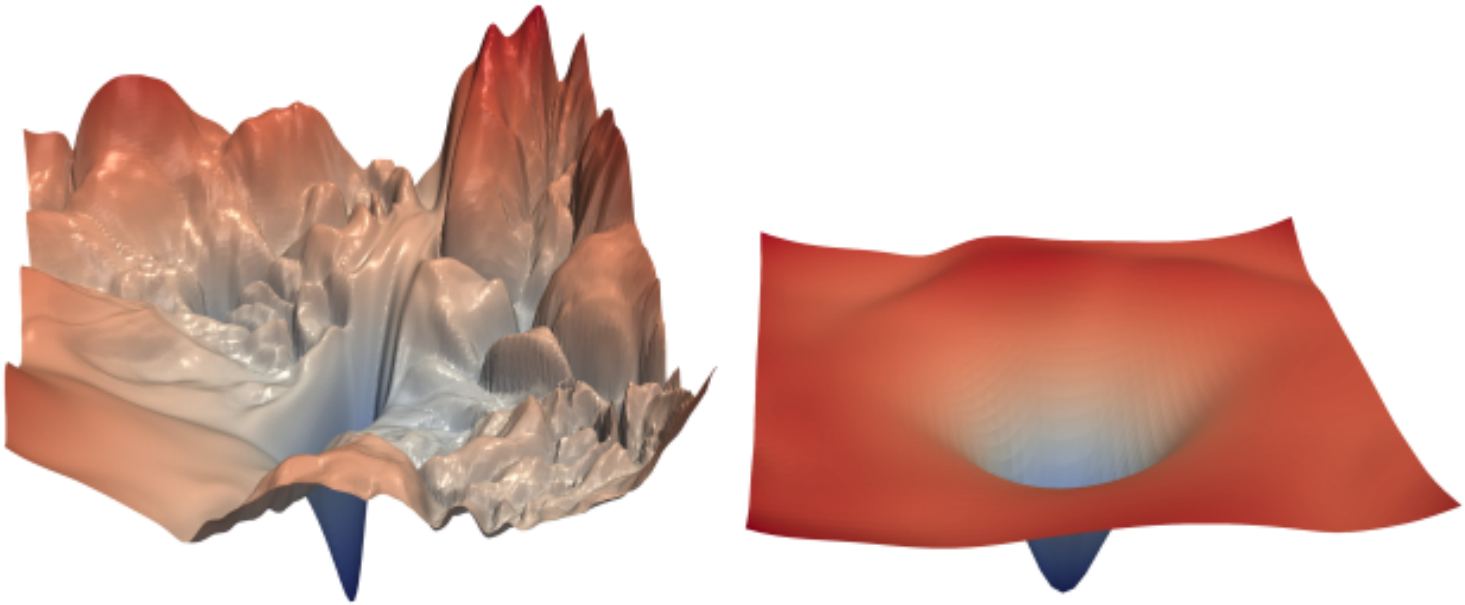
- Propriedade:
  - Não negativa.
  - Linear.
  - Robusto.
  - Convexa.
  - Não é diferenciável.

Função de perda de Huber:

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

δ hiper parâmetro.

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



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

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: $p(y = 1|\boldsymbol{x})$ e $p(y = 0|\boldsymbol{x})$.
  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.

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

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

- Perda categórica de entropia cruzada.

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

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.

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

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

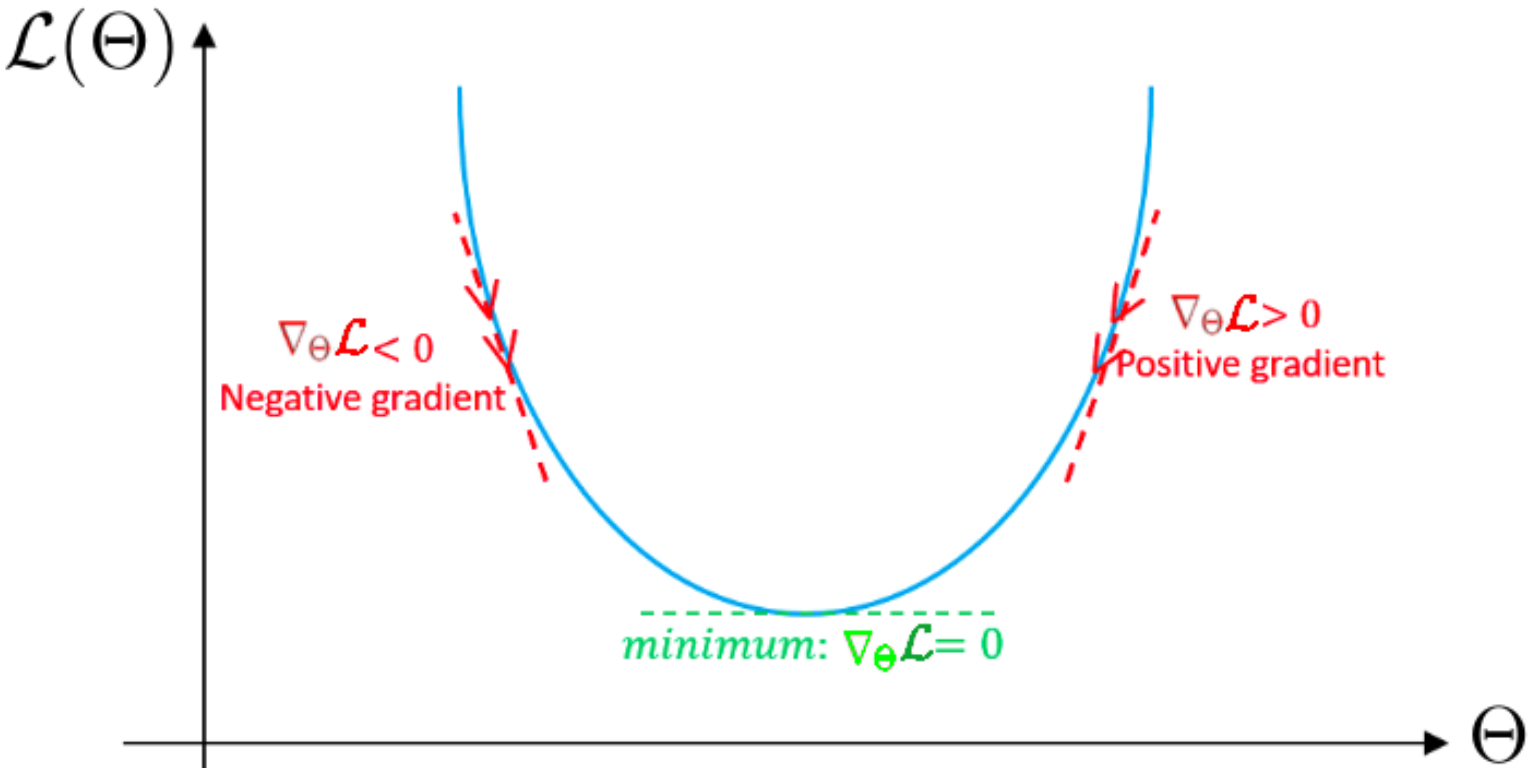
- Queremos obter Θ que minimize J, ou seja, $\min_{\omega,b} \mathcal{L}(\boldsymbol{\omega}, b)$
  - n $\Rightarrow$ O mínimo global.
- Podemos determinar o gradiente descendente como:

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

- Algoritmo:

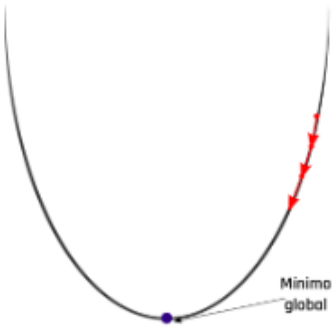
$$\begin{aligned} \boldsymbol{\omega} &=: \boldsymbol{\omega} - \eta \frac{\partial}{\partial \boldsymbol{\omega}} \mathcal{L}(\omega, b) \\ b &=: b - \eta \frac{\partial}{\partial b} \mathcal{L}(\omega, b) \end{aligned} \tag{15}$$

onde η é a taxa de aprendizado.

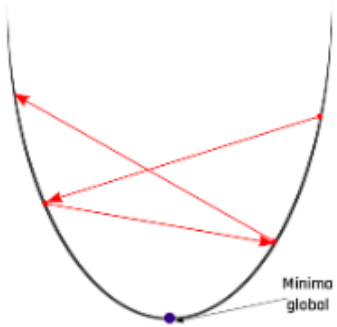

$\mathcal{L}(\Theta)$
$\nabla_\Theta \mathcal{L} < 0$
Negative gradient
$\nabla_\Theta \mathcal{L} > 0$
Positive gradient
minimum: $\nabla_\Theta \mathcal{L} = 0$
Θ

Taxa de aprendizado

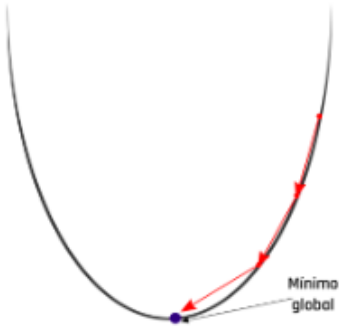
- Taxa de aprendizado:
  - η é muito pequeno, gradiente descendente pode ser muito lento.
  - η muito grande pode:
    1. Nunca alcança o mínimo.
    2. Falha para convergir



Taxa de aprendizado pequena



Taxa de aprendizado grande

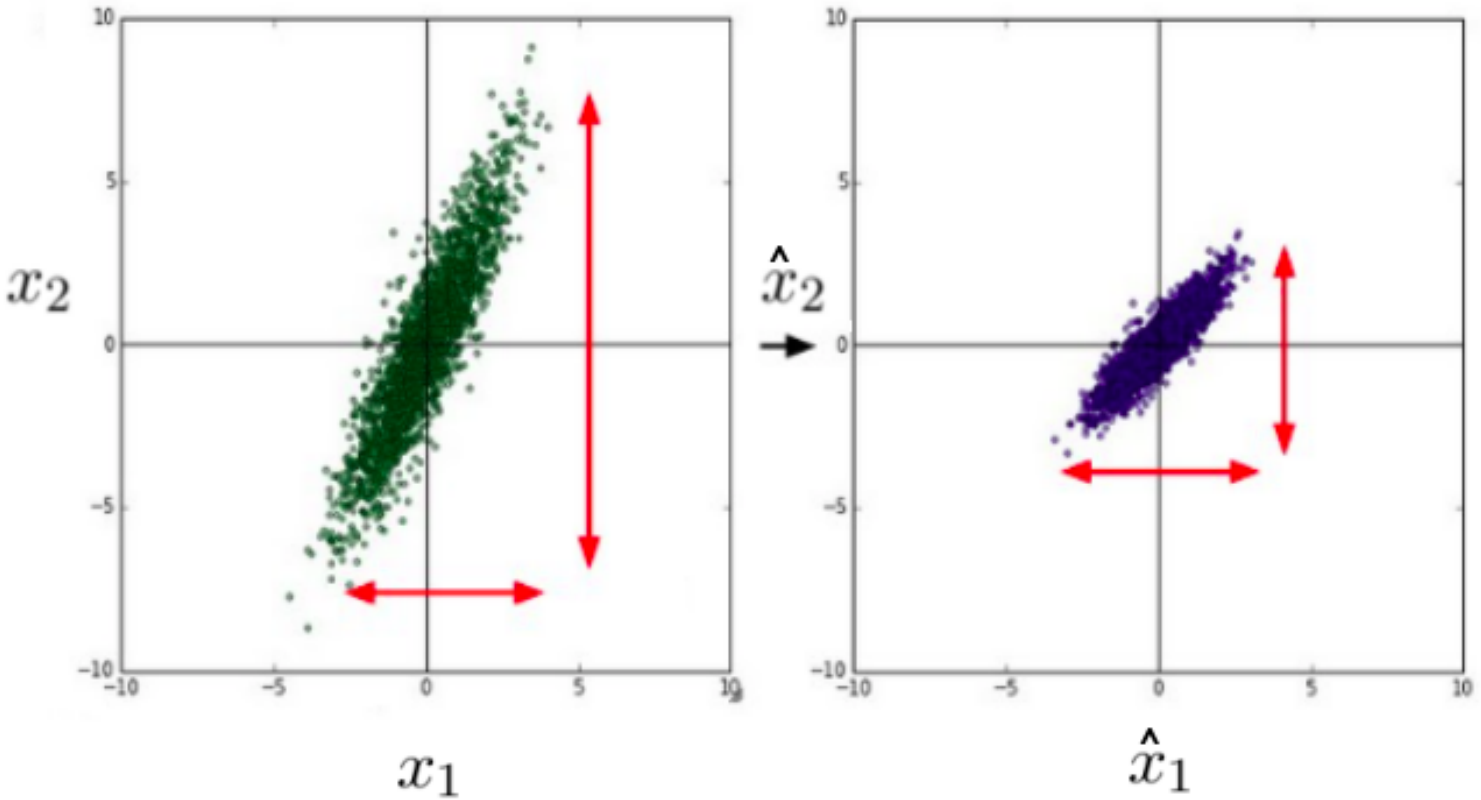


Taxa de aprendizado ideal

- Diferença entre gradiente descendente e gradiente descendente estocástico.

Código: Gradiente.

Normalizando as características



$$f(x) = \omega_1 x_1 + \omega_2 x_2 + b \tag{16}$$

onde $x_1 >> x_2$.

- 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 μ 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 μ é a média e σ é o desvio padrão.

General Learning Principle

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

"On-line" mode

1. Initialize $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
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 $\mathbf{w} := \mathbf{0} \in \mathbb{R}^m$, $\mathbf{b} := 0$
2. For every training epoch:
   A. Initialize $\Delta\mathbf{w} := 0,\ \Delta b := 0$
   B. For every $\langle \mathbf{x}^{[i]}, y^{[i]} \rangle \in \mathcal{D}$:
      (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 $\mathcal{D} = (\langle \mathbf{x}^{[1]}, y^{[1]} \rangle, \langle \mathbf{x}^{[2]}, y^{[2]} \rangle, ..., \langle \mathbf{x}^{[n]}, y^{[n]} \rangle) \in (\mathbb{R}^m \times \{0,1\})^n$

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 $\{\langle \mathbf{x}^{[i]}, y^{[i]} \rangle, ..., \langle \mathbf{x}^{[i+k]}, y^{[i+k]} \rangle\} \subset \mathcal{D}$:
      (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

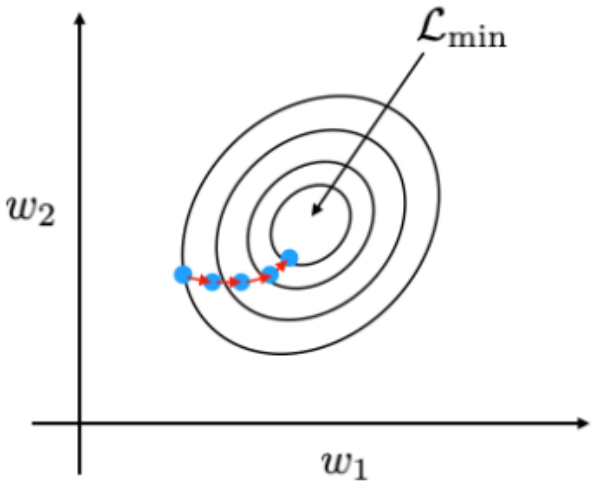
Perceptron learning rule

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

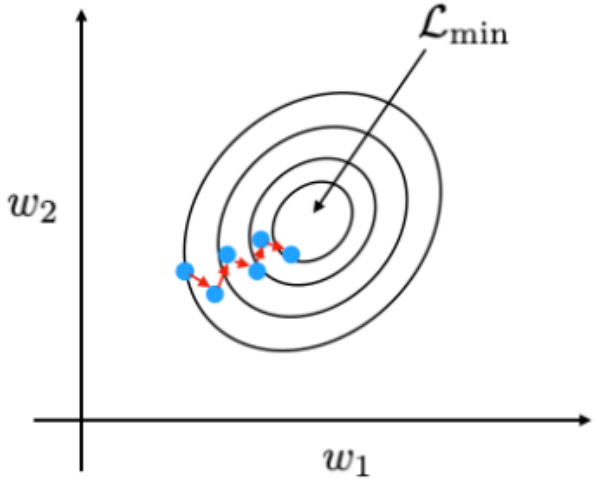
Stochastic gradient descent

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) $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]T}\mathbf{w} + b\right)$
      (b) $\nabla_{\mathbf{w}}\mathcal{L} = \left(y^{[i]} - \hat{y}^{[i]}\right)\mathbf{x}^{[i]}$
          $\nabla_b\mathcal{L} = \left(y^{[i]} - \hat{y}^{[i]}\right)$
      (c) $\mathbf{w} := \mathbf{w} + \eta \times (-\nabla_{\mathbf{w}}\mathcal{L})$
          $b := b + \eta \times \underbrace{(-\nabla_b\mathcal{L})}_{\text{negative gradient}}$

learning rate



minibatch



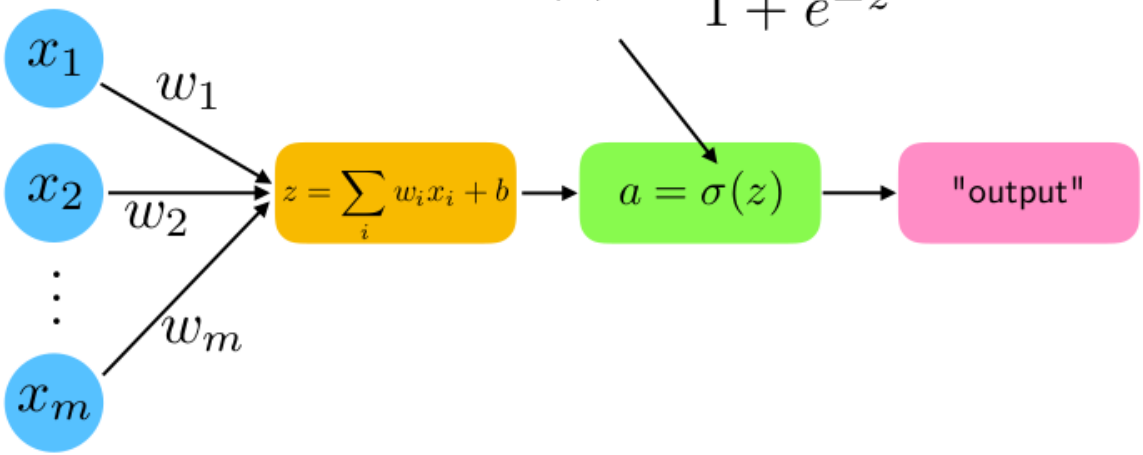
on-line

Multi-class Classification

Logistic Regression Neuron

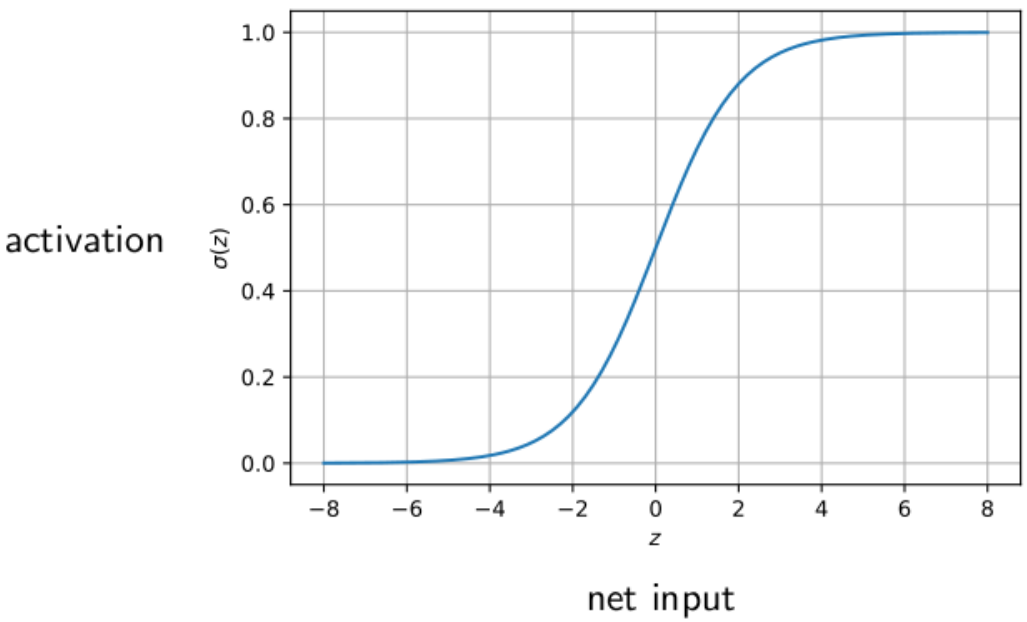
For binary classes $y \in \{0, 1\}$

"logistic sigmoid" $\sigma(z) = \dfrac{1}{1 + e^{-z}}$



Logistic Sigmoid Function

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



Log-Likelihood "Loss"

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

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

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

Negative Log-Likelihood Loss

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

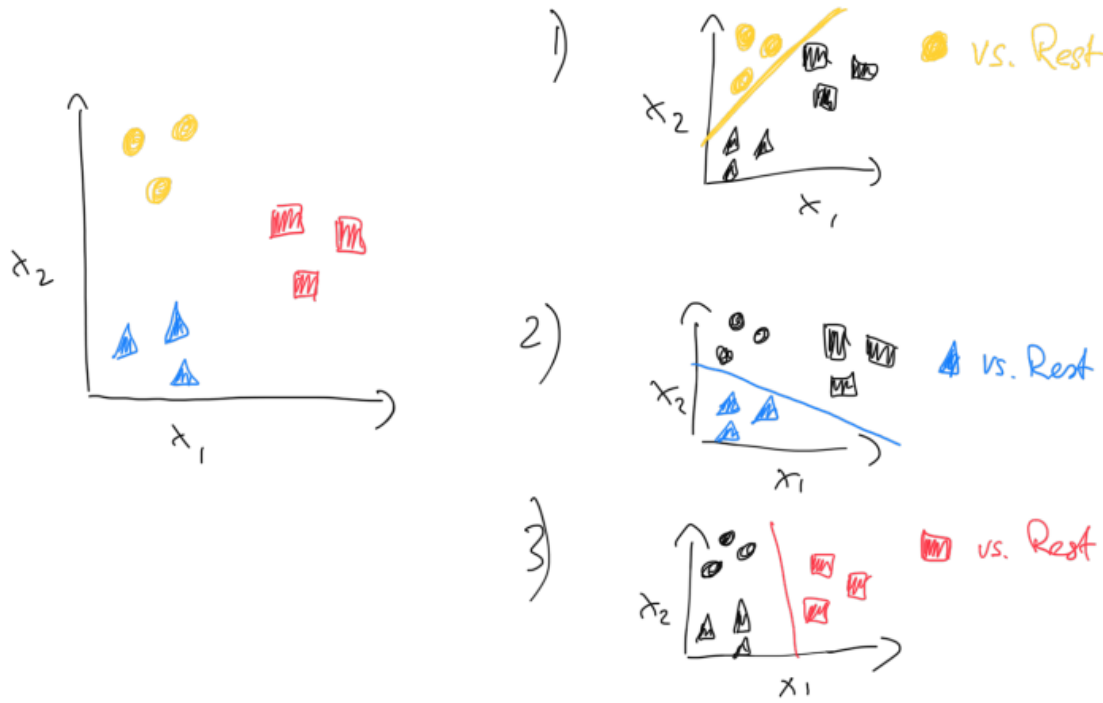
$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= -l(\mathbf{w}) \\ &= -\sum_{i=1}^{n} \left[y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right) \right]\end{aligned}$$

(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)

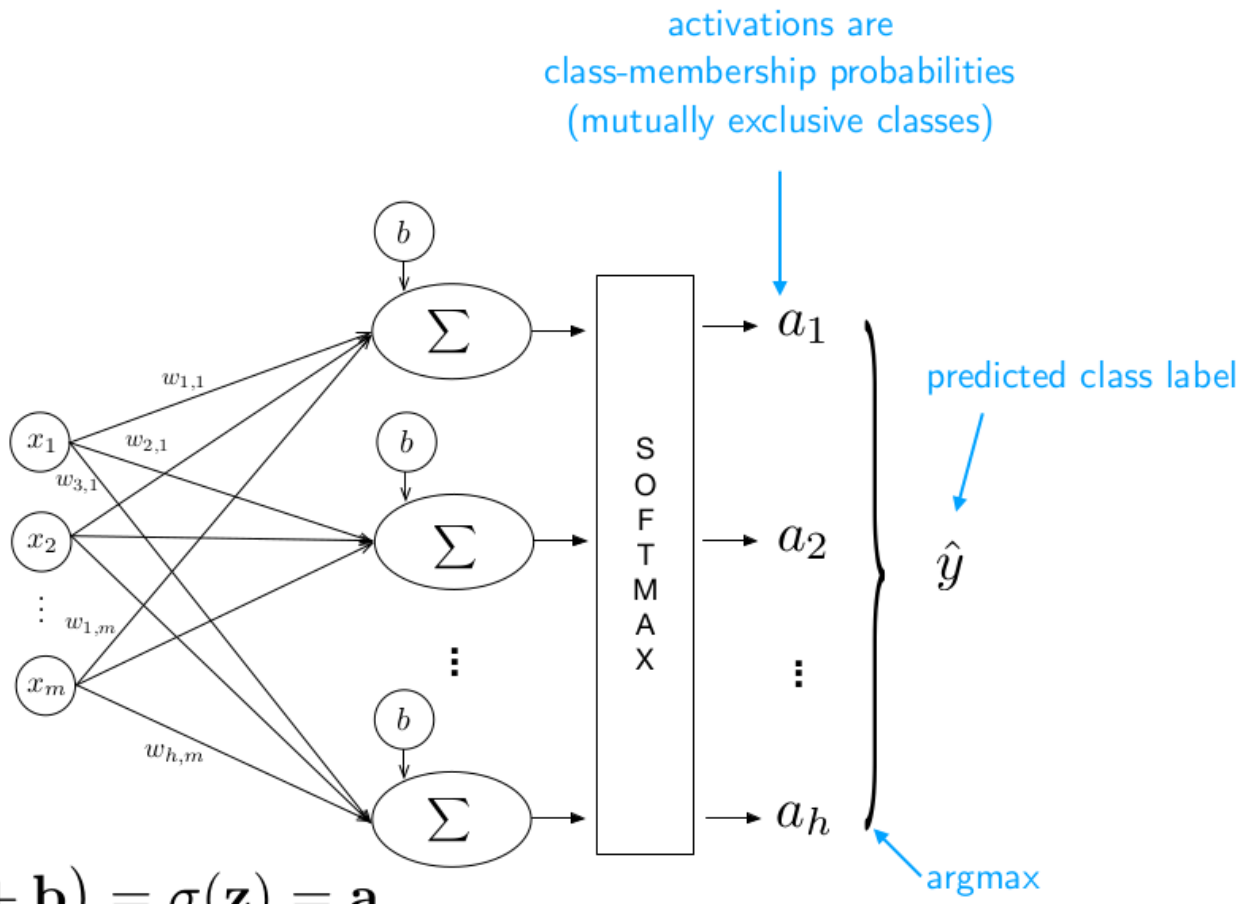
Softmax Regression

Multi-Class Classification with Multiple Binary Classifiers

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



Multinomial Logistic Regression / Softmax Regression



$$\sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) = \sigma(\mathbf{z}) = \mathbf{a}$$

Softmax Activation

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

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

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!

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

(Multi-category) Cross Entropy for k different class labels

Cross Entropy Loss Function Example

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

(4 training examples, 3 classes)

Cross Entropy Loss Function Example

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

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

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

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

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

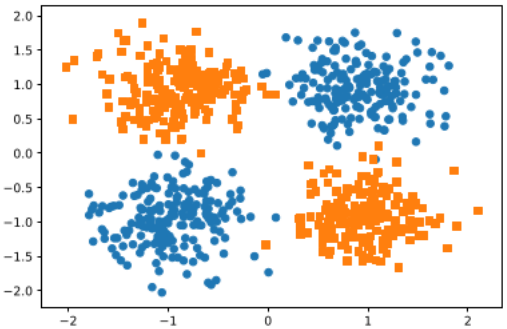
$$\mathcal{L}_{\text{multiclass}} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} -y_j^{[i]} \log\left(a_k^{[i]}\right)$$

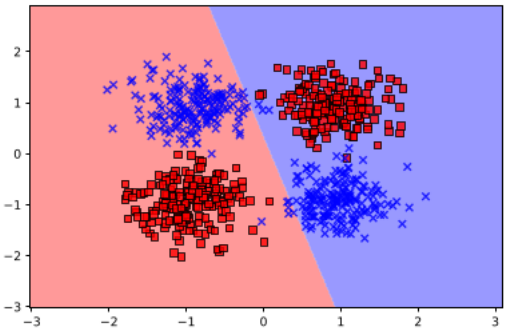
$$\approx 0.9335$$

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

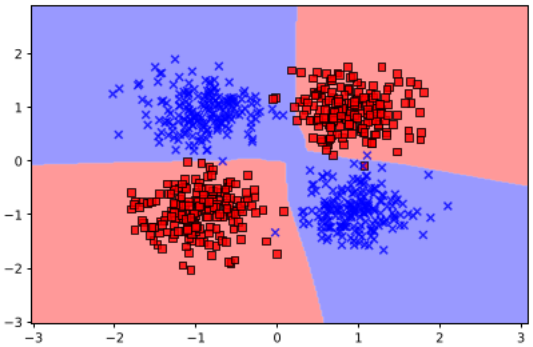
Multilayer Perceptrons

- Problema de superfícies não lineares.





1-hidden layer MLP
with linear activation function

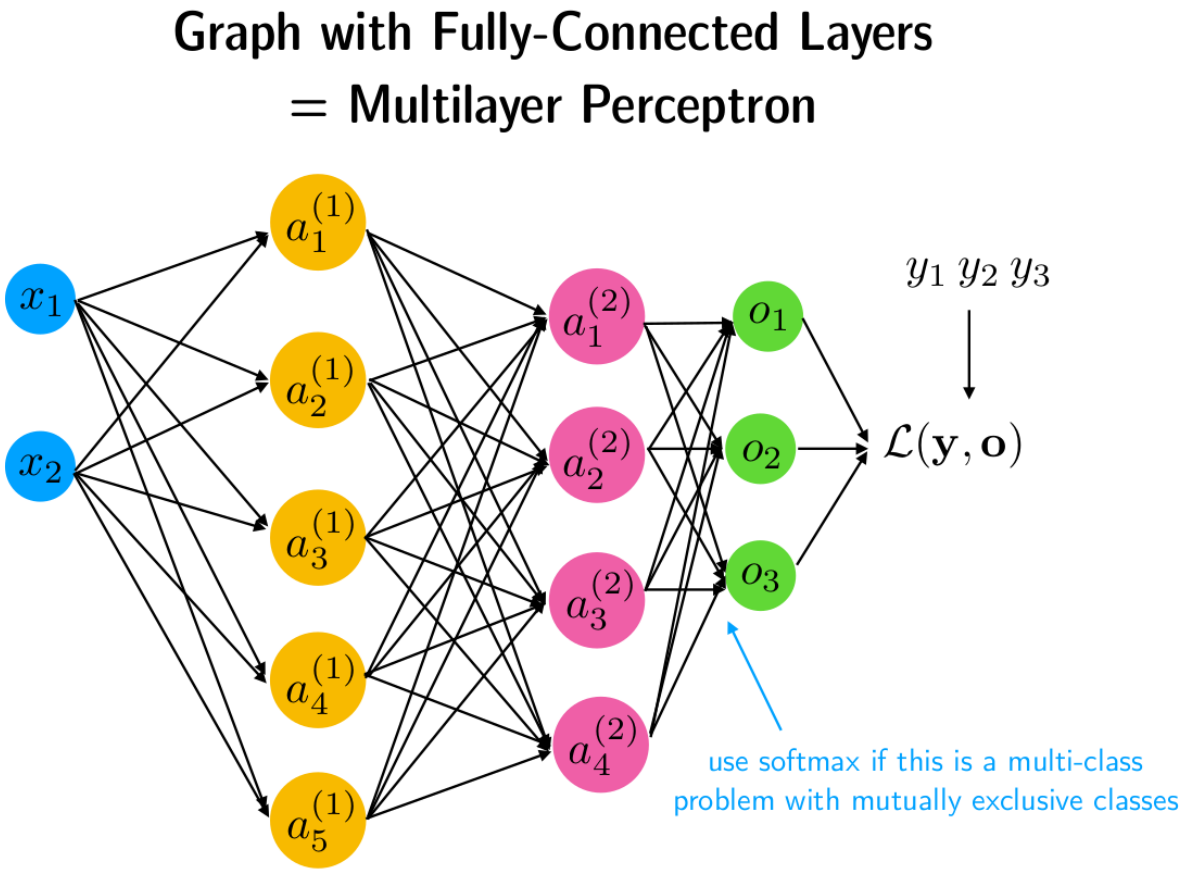


1-hidden layer MLP
with non-linear activation function (ReLU)

Multilayer Perceptrons

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

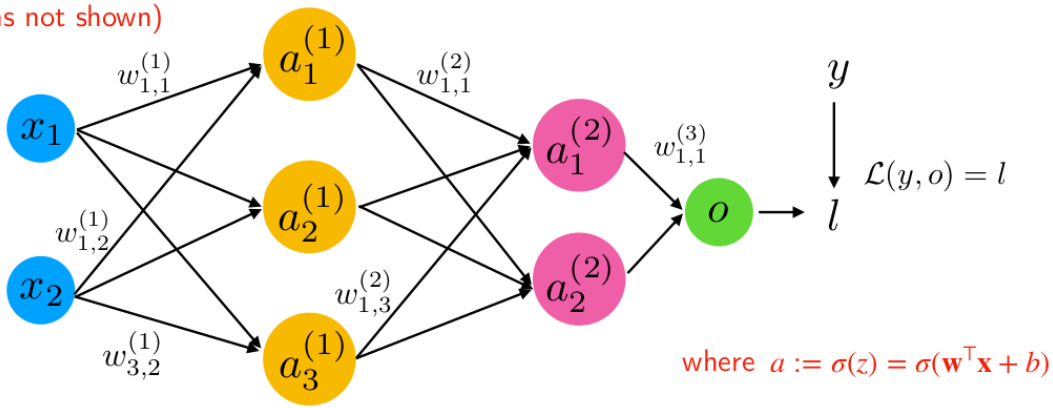
Multilayer Perceptrons



Graph with Fully-Connected Layers = Multilayer Perceptron

Nothing new, really

(bias not shown)



where $a := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

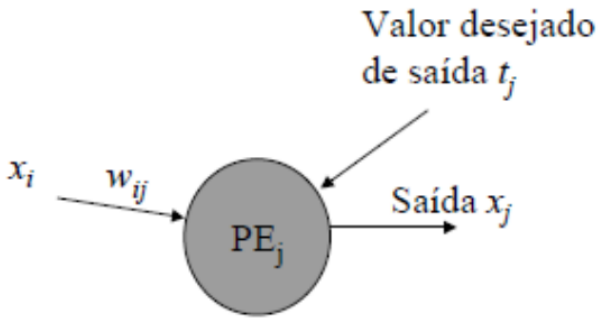
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

(Assume network for binary classification)

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:

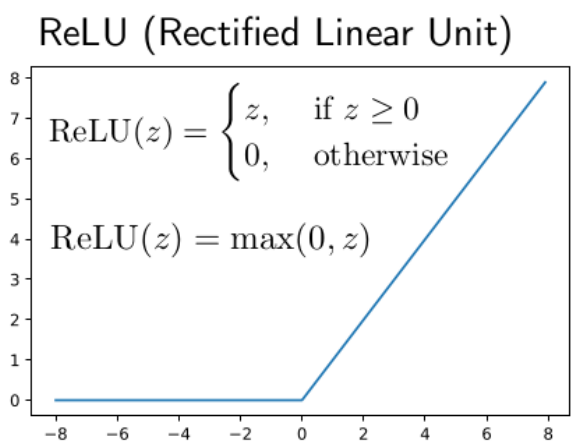


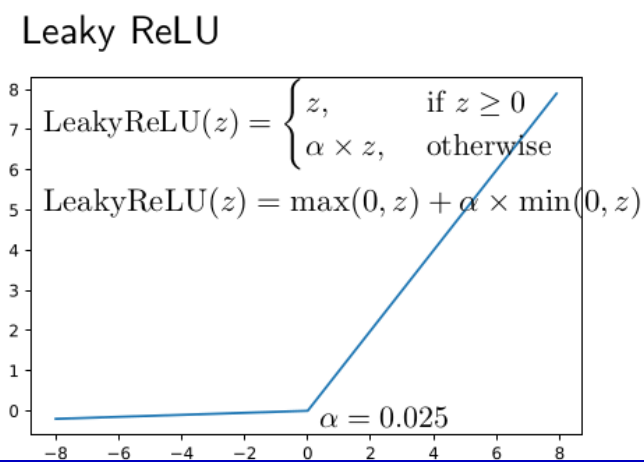
$$e_j = (t_j - x_j) F'(y_j)$$

Material: L08 mlp slides extras

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.

Codigo: Softmax

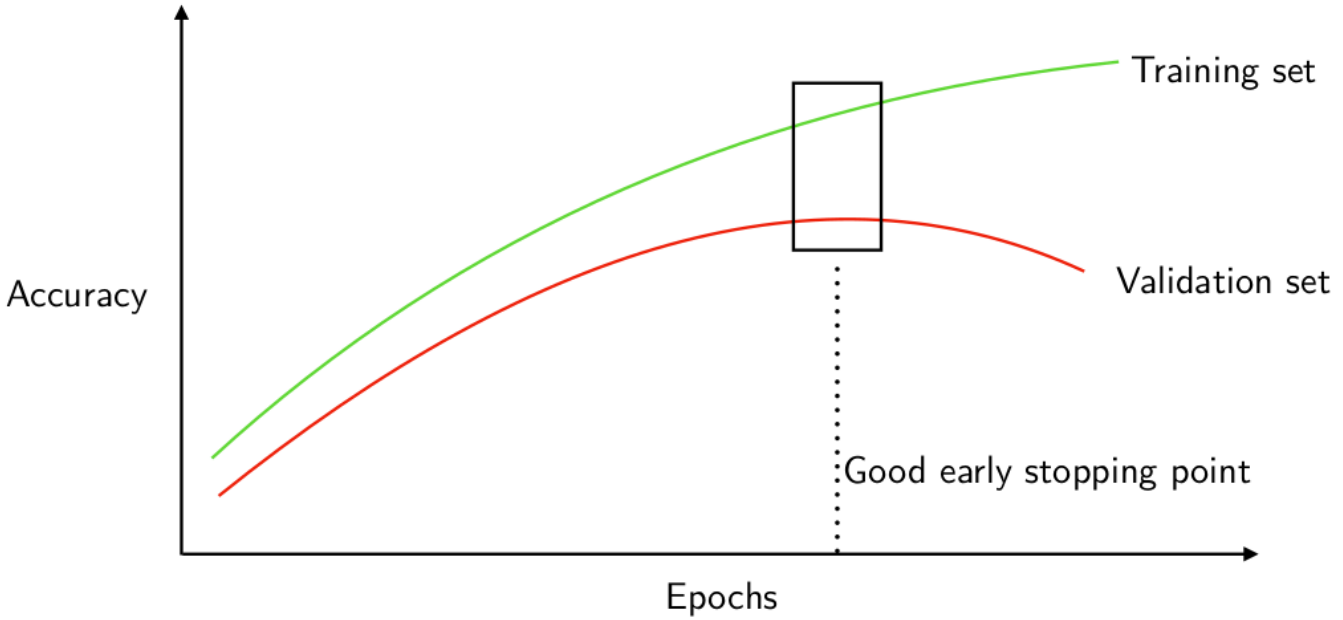
1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

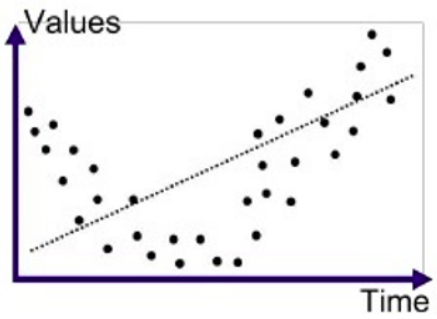
Regularization

Sebastian Raschka STAT 453: Intro to Deep Learning and Generative Models SS 2020 1

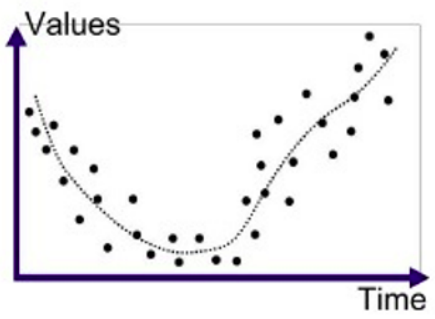
Step 2: Early stopping (not very common anymore)

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

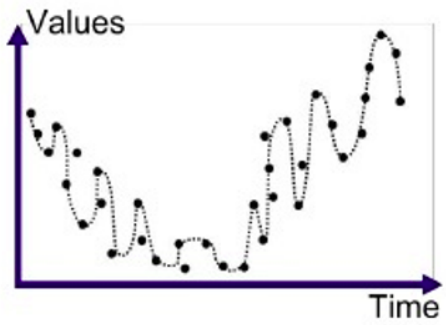




Underfitted



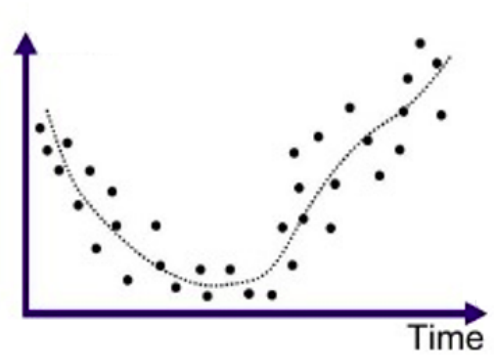
Good Fit/Robust



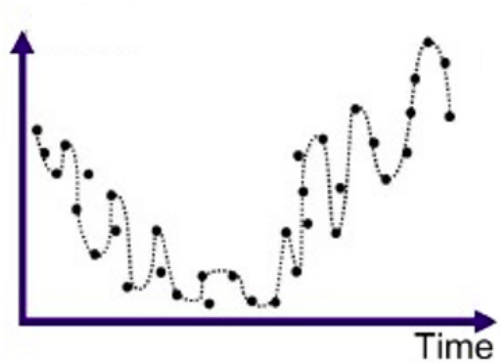
Overfitted

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.
  - Drupout
- Regularização permite manter todas as características, mas evita ques 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 ω_3, ω_4 pequeno o suficiente (≈ 0).
- Isso é chamado de regularização ou penalização de ω_3 e ω_4.

- Nem sempre podemos saber quais pesos ω_j podemos penalizar, assim em geral, regularizamos todos ω_j

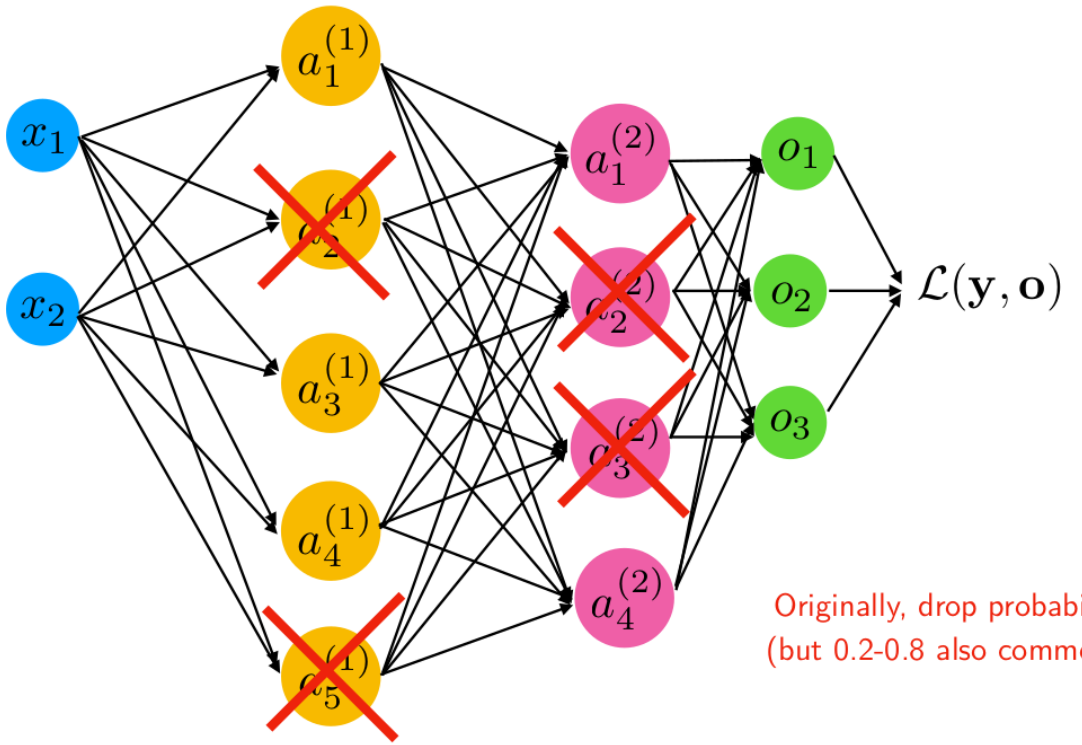
$$\min_{\boldsymbol{\omega},b} \mathcal{L}(\boldsymbol{\omega}, b) = \min_{\boldsymbol{\omega},b} \left[\frac{1}{2n} \sum_{i=1}^{n} \left(f_{\omega,b}(x^i) - y^i \right)^2 + \frac{\lambda}{2n} \sum_{j=1}^{n} \omega_j^2 \right] \quad (17)$$

onde λ é um hiperparãmetro regularizador.

- Se escolhermos $\lambda \approx 0$, não há regularização e não irá impedir o overfit.
- Se escolhermos um valor muito grande ($\lambda \approx 10^1 0$), 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



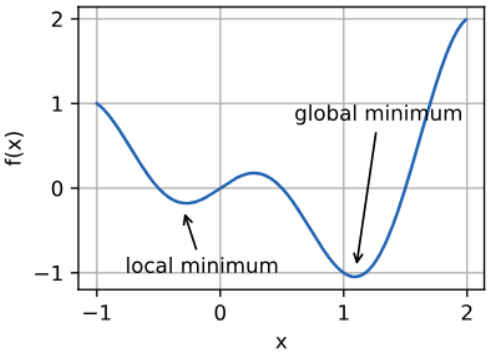
Por que o Dropout funciona bem?

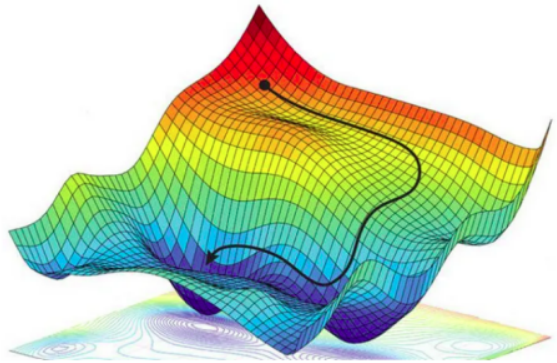
- A rede aprenderá a não depender também de conexões específicas fortemente
- Assim, considerará mais conexões (porque não pode contar com individuais)
- Os valores de peso serão mais dispersos (pode levar a menores pesos como com a norma L2)
- Nota lateral: Você certamente pode usar diferentes probabilidades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

Codigo: Regularization Dropout MLP Keras

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

Optimização





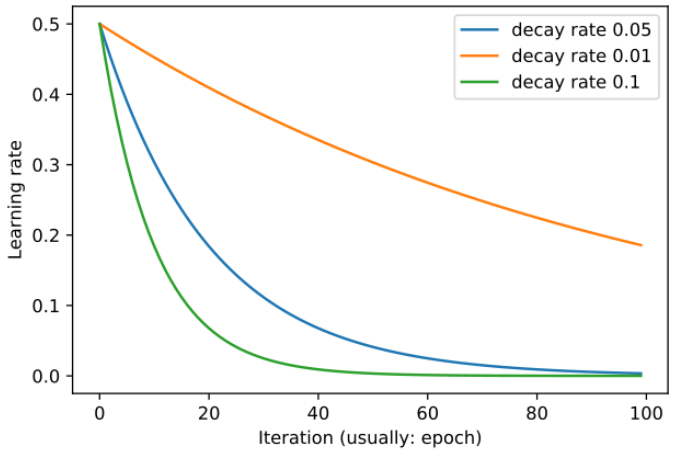
Learning Rate Decay

Most common variants for learning rate decay:

1) Exponential Decay:

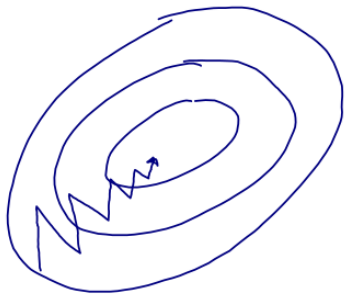
$\eta_t := \eta_0 \cdot e^{-k \cdot t}$

where k is the decay rate

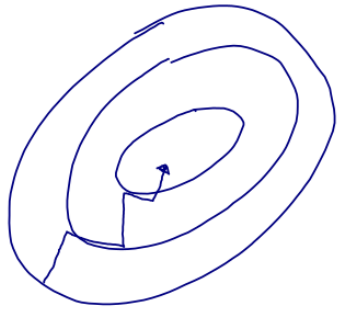


Training with "Momentum"

- 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"

Often referred to as "velocity" v

"velocity" from the previous iteration

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

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

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:

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

m_{t-1} m_t original momentum term

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

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:

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

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

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

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:

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

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:

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

codigo: ode mlp

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

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

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

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 $u_0 : \mathcal{D} \to \mathbb{R}$ 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

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

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

The method constructs a neural network approximation

$$u_\theta(t,x) \approx u(t,x) \tag{22}$$

of the solution of nonlinear PDE, where $u_\theta : [0,T] \times \mathcal{D} \to \mathbb{R}$ denotes a function realized by a neural network with parameters θ.
The continuous time approach for the parabolic PDE is based on the (strong) residual of a given neural network approximation $u_\theta \colon [0,T] \times \mathcal{D} \to \mathbb{R}$ of the solution u, i.e.,

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

To incorporate this PDE residual r_θ 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_θ 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

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

where X denotes the collection of training data and the loss function ϕ_θ contains the following terms:
- the mean squared residual

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

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_θ 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_θ is the neural network approximation of the solution $u\colon [0, T] \times \mathcal{D} \to \mathbb{R}$.

Note that the training data X consists entirely of time-space coordinates.
Solving the problem with Stochastic Gradient Descent:

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

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 $\mathcal{L}(\boldsymbol{\theta};\mathcal{T})$.

Codigo: 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.
- n 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!

- $\mathcal{F}$ => 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 ε:

$$\begin{aligned} \varepsilon &= \|\hat{u}_T - u\| \leq \|\hat{u}_T - u_T\| + \|u_T - u_{\mathcal{F}}\| + \|u_{\mathcal{F}} - u\| \\ \varepsilon &= \varepsilon_{opt} + \varepsilon_{gen} + \varepsilon_{app} \end{aligned}$$

- $\varepsilon_{app} => u_{\mathcal{F}} \longrightarrow u$
- ε_{gen} Mede a capacidade de generalização da rede, quando o erro de generalização domína, temos o "overfitting".
- ε_{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.

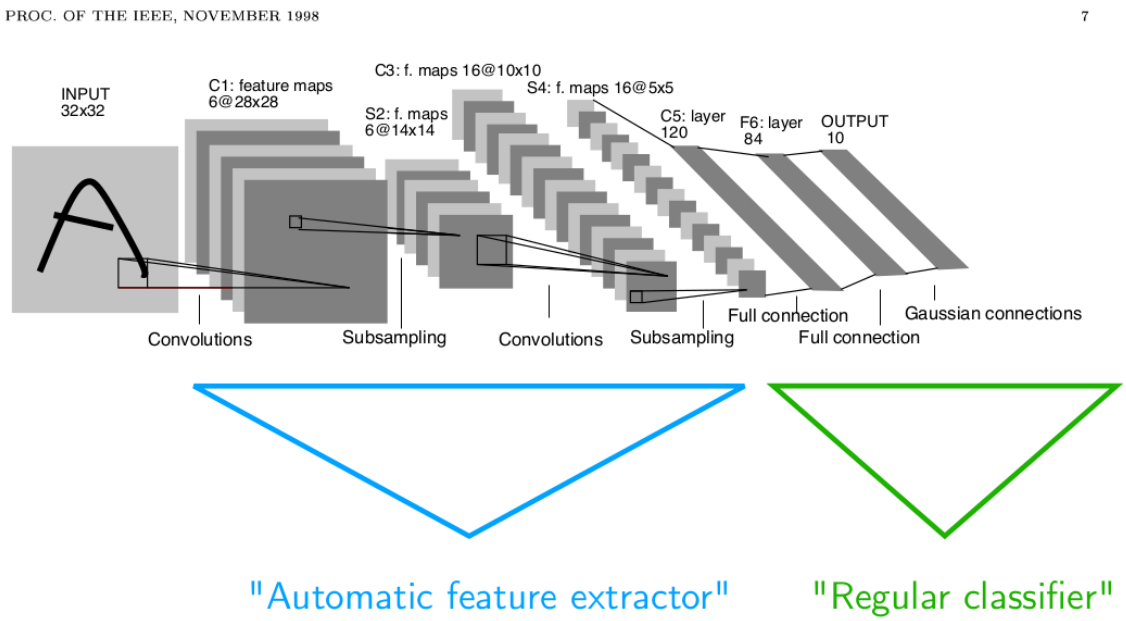
1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs

10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

Redes Convolucionais

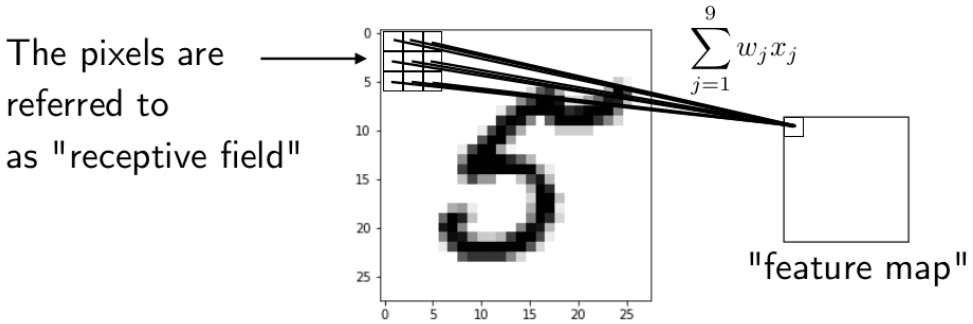
- Compartilhamento de pesos (menor custo computacional).
- Ótimo estrator de características.
- RevOlução na visão computacional.

Hidden Layers



Weight Sharing

A "feature detector" (filter, kernel) slides over the inputs to generate a feature map

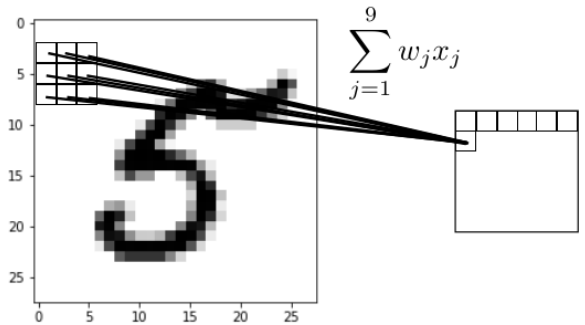


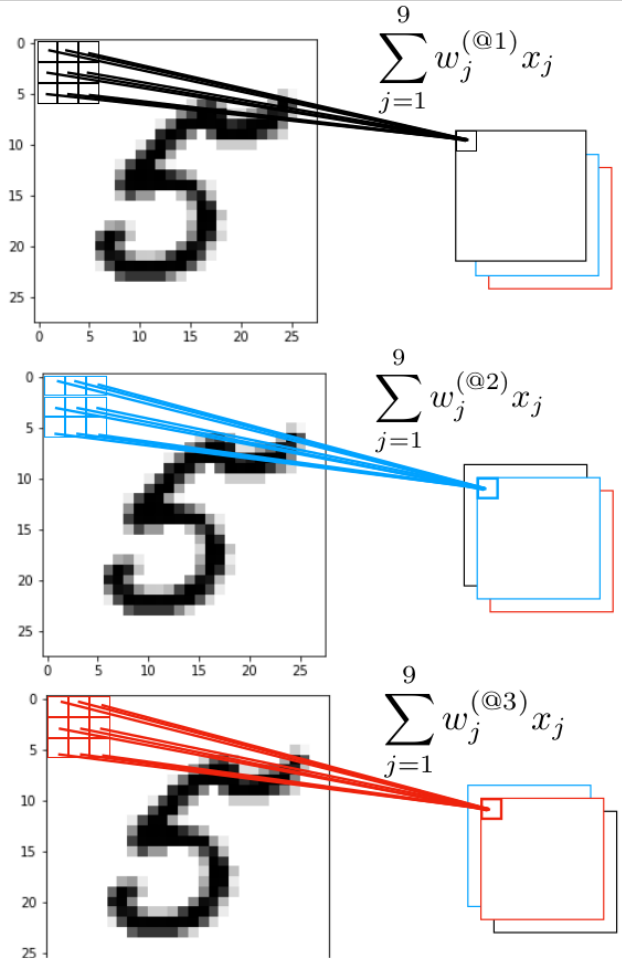
Rationale: A feature detector that works well in one region may also work well in another region

Plus, it is a nice reduction in parameters to fit

Weight Sharing

A "feature detector" (kernel) slides over the inputs to generate a feature map



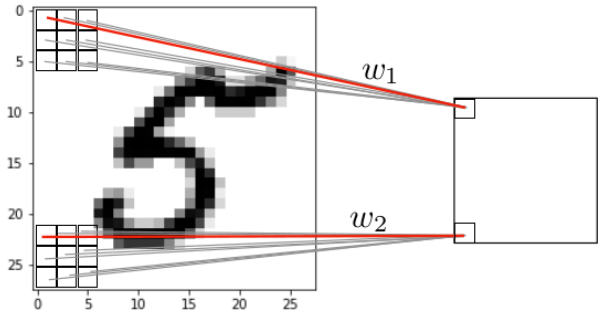


Multiple "feature detectors" (kernels) are used to create multiple feature maps

Backpropagation in CNNs

Same overall concept as before: Multivariable chain rule, but now with an additional weight sharing constraint

Due to weight sharing: $w_1 = w_2$

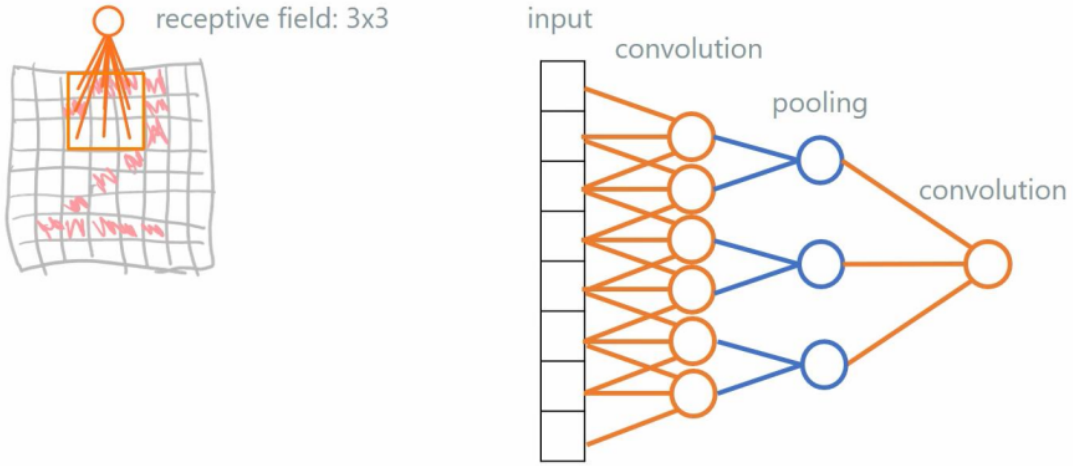


Optional averaging

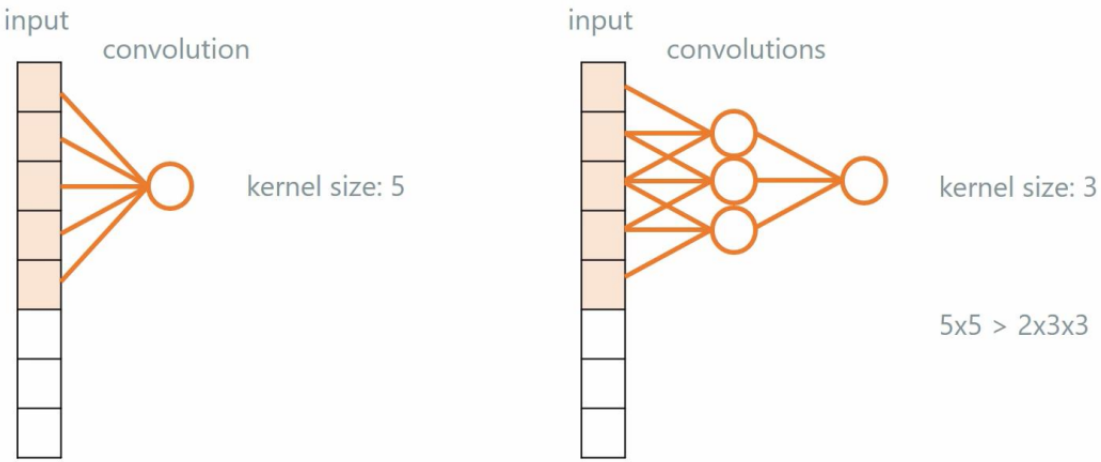
weight update:

$$w_1 := w_2 := w_1 - \eta \cdot \frac{1}{2}\left(\frac{\partial \mathcal{L}}{\partial w_1} + \frac{\partial \mathcal{L}}{\partial w_2}\right)$$

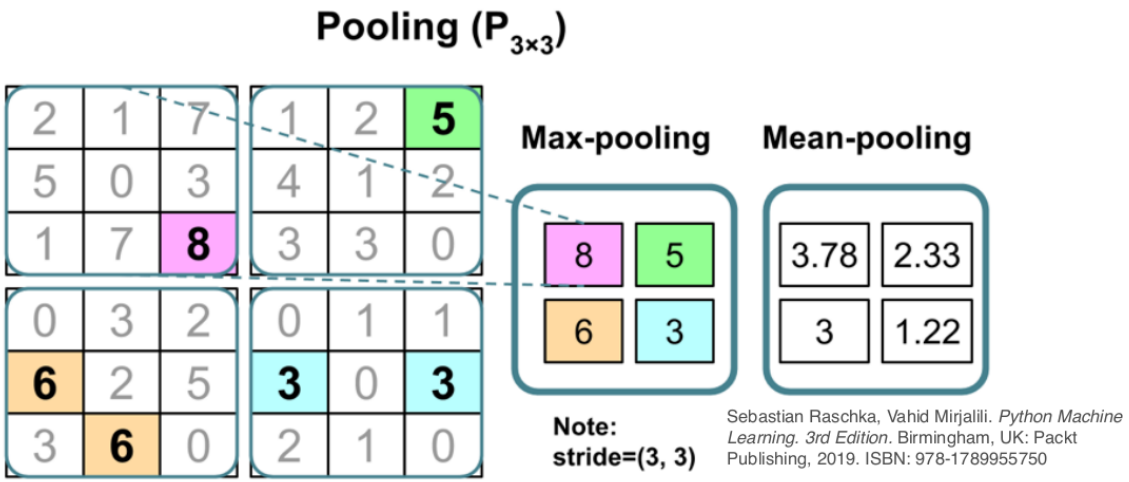
Receptive field



Kernel size


input
convolution
kernel size: 5
input
convolutions
kernel size: 3
5x5 > 2x3x3

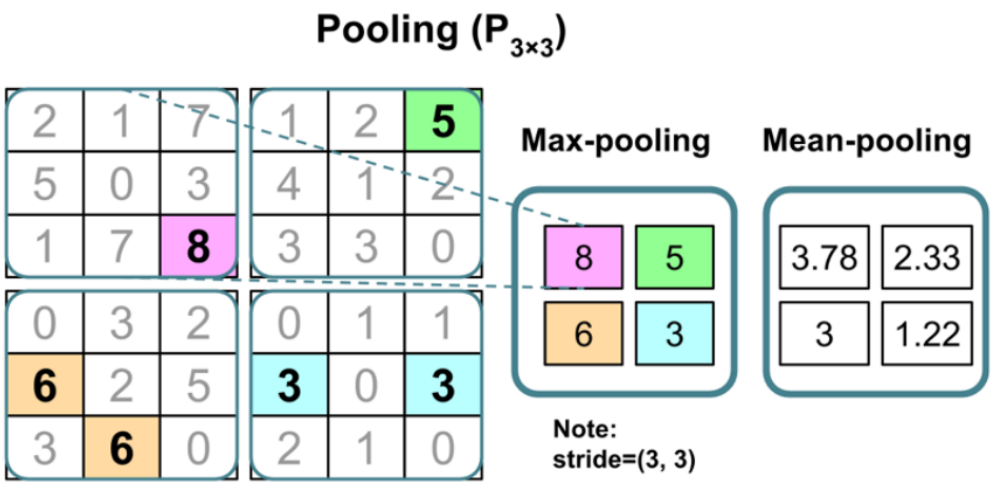
Pooling Layers Can Help With Local Invariance



Sebastian Raschka, Vahid Mirjalili. *Python Machine Learning. 3rd Edition*. Birmingham, UK: Packt Publishing, 2019. ISBN: 978-1789955750

Downside: Information is lost.

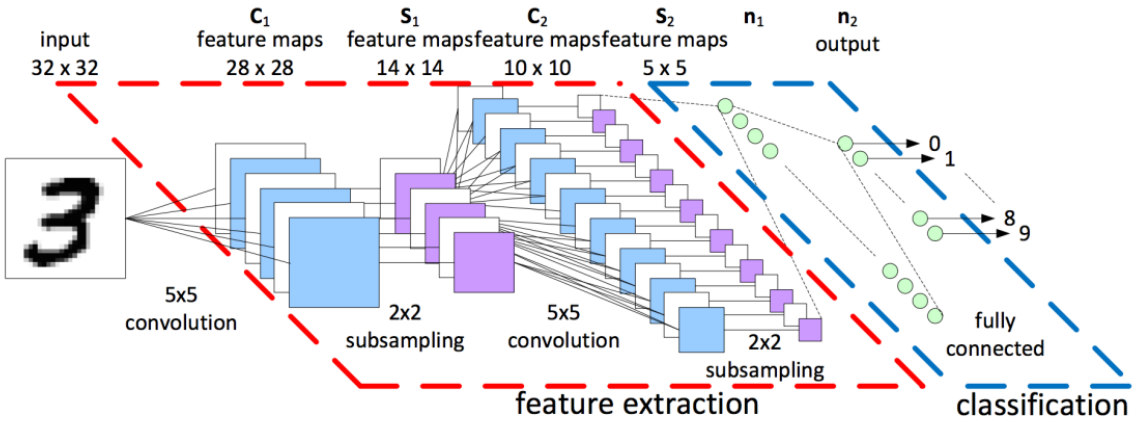
Pooling Layers Can Help With Local Invariance

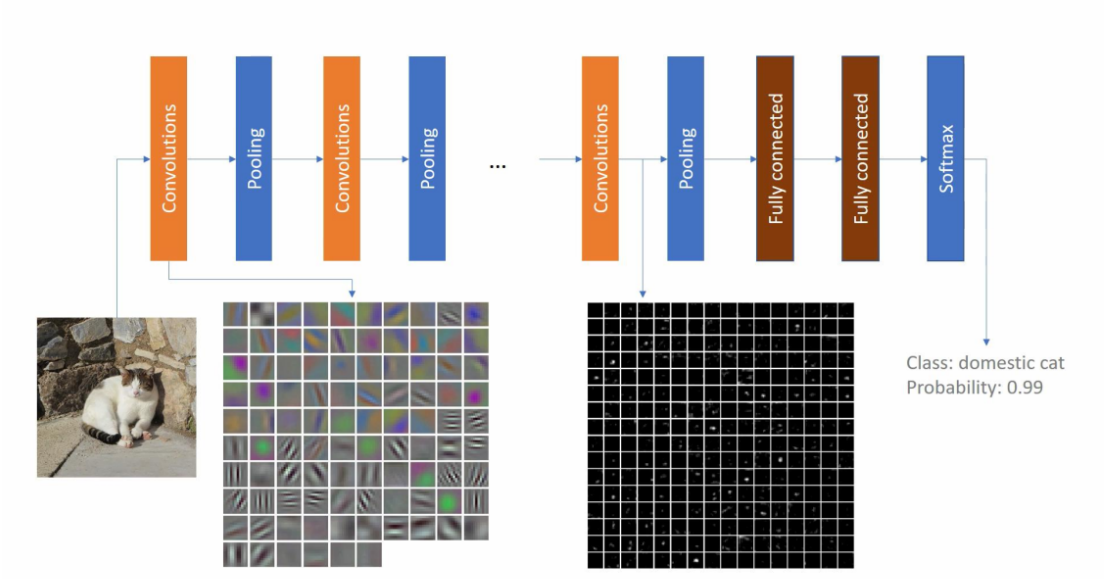


Note that typical pooling layers do not have any learnable parameters

Downside: Information is lost.

May not matter for classification, but applications where relative position is important (like face recognition)


input
32 x 32
C1
feature maps
28 x 28
S1
feature maps
14 x 14
C2
feature maps
10 x 10
S2
feature maps
5 x 5
n1
n2
output
0
1
8
9
5x5
convolution
2x2
subsampling
5x5
convolution
2x2
subsampling
fully
connected
feature extraction
classification



https://miro.medium.com/v2/resize:fit:526/1*GcI7G-JLAQiEoCON7xFbhg.gif Exemplo código

- Arquiteturas:
  - AlexNet
  - VGG
  - GoogleNet
  - ResNet
- O aprendizado por transferência, usado no aprendizado de máquina, é a reutilização de um modelo pré-treinado em um novo problema. Na aprendizagem por transferência, uma máquina explora o conhecimento adquirido em uma tarefa anterior para melhorar a generalização sobre outra.

1 Bibliotecas
2 Introduction
3 Supervised learning
4 Deep Learning
5 Perceptron
6 Multilayer Perceptrons
7 Regularization
8 Optimização
9 Solving differential equations using neural networks
- Using physics informed neural networks (PINNs) to solve parabolic PDEs
10 Redes Convolucionais
11 Operador Neural
- Formulação Kernel
- Operador Neural de Fourier

Operador Neural

- Matematicamente operadores são usualmente referidos como mapeadores entre espaço de funções
  - Integração
  - Diferenciação
- Diferente dos métodos clássicos de solucionadores de equações diferenciais como, por exemplo, elementos finitos, diferença finita que discretizam o domínio físico e tornam o problema no espaço euclidiano de dimensão finita, o operador neural é capaz de aprender diretamente o operador da equação diferencial e são invariantes a discretização do domínio, que em certo sentido, está mais próximo da solução real, ou seja, aprende a função continua ao invés de valores discretizados.

- São independente das malhas de discretização, que produz um conjunto de parâmetros que podem ser usado em diferentes tipos de discretização.
  - Transfere soluções entre malhas diferentes.
- São treinados diretamente nos dados.
- Aprendem o mapeamento de funções, generalizando em diferentes discretizações.
  - Podem ser treinados com dados menos precisos e ainda são capazes de aprenderem e gerarem resultados bem precisos.

Considere o domínio $\mathcal{D} \in \mathcal{R}^d$.
Considere a equação diferencial:

$$\mathcal{L}u = f$$

onde $\mathcal{L}$ é um operador que mapea u para uma função f.
Por exemplo:

$$\begin{aligned} -\nabla \cdot (a(x)\nabla u(x)) &= f(x) \qquad x \in D \\ u(x) &= 0 \qquad x \in \partial D, \end{aligned}$$

- Para qualquer função $a(x)$, a equação tem uma solução $u(x)$, mantendo $f(x)$ fixo.
  $\mathcal{A}$ é o espaço de funções de $a(x)$ a ser mapeado no espaço de funções $\mathcal{U}$.
  $\mathcal{A} \Rightarrow \mathcal{G}^{\dagger} \Rightarrow \mathcal{U}$
  - $a_j(x)$ => Função no espaço $\mathcal{A}$
  - $u_j(x)$ => Função no espaço $\mathcal{U}$
  - $\{a_j, u_j\}$ são os pares de entrada-saída.

  . Queremos aprender o operador $\mathcal{G}^{\dagger}$ que faz esse mapeamento.
  - $u_j = \mathcal{G}^{\dagger}(a_j)$
  - $\mathcal{G}_{\theta} \sim \mathcal{G}^{\dagger}$

  onde $\mathcal{G}_{\theta}$ é uma rede neural com parâmetro θ (aprendido).

- Queremos oncontrar θ que minimise a função de custo.

$$\min_{\theta} \mathcal{L}_{a\sim u}[\mathcal{G}(a,\theta), \mathcal{G}_a^{\dagger}] \tag{26}$$

$a(x)$ e $u(x)$ são funções. Para o processo de treinamento vamos trabalhar com $a(x), u(x)$ como sendo dados, ou seja essas funções são discretizadas em a_j, u_j. Essa discretização pode ser feita por qualquer método. Com isso podemos usar como dados de treinamento os pares $\{a_j, u_j\}_{j=1}^{N}$. (N é o número de pontos discretizados.)

- Resumindo: $a(x), u(x)$ vem como dados discretizados ($P_N = (\boldsymbol{x}_1, \cdots \boldsymbol{x}_N) \quad \in \quad D$) e $\{a_j, u_j\}$ são observações.

Formulação Kernel

$$
\begin{aligned}
(\zeta_a u)(x) = f(x) \qquad & x \in D \\
u(x) = 0 \qquad & x \in \partial D,)
\end{aligned}
$$

Sob condições gerais, podemos definir a função de Green (G) como a única solução do problema.

$$\zeta_a G(x) = \delta_x \tag{27}$$

δ_x é a delta de Dirac, o que torna a equação homogênea, pois $\delta_x = 0$ para $x \neq y$.

$$u(x) = \int_D G_a(x, y) f(y) dy$$

Como $x \neq y$ é contínua, então ζ_a é uniformemente contínua. Podemos modelar o kernel atravéz de uma rede neural $\mathcal{K}$, semelhante a função de Green. A rede $\mathcal{K}$ recebe uma entrada (x, y). Como $\mathcal{K}$ depende de a, vamos deixar que $\mathcal{K}$ dependa da entrada $(a(x), u(x))$

$$u(x) = \int_D \mathcal{K}(x, y, a(x), a(y)) f(y) dy$$

Vamos discretizar $u(x)$ como sendo $u(x) \sim u(x)_t$, onde u é iterado no tempo $t(t = 0, \cdots, .T)$.

$$u_{t+1} = \int_D \mathcal{K}(x, y, a(x), a(y))u_t dy \tag{28}$$

onde $u_0(x) = (x, a(x))$ e u_T é a saída final.
Para usar a vantagem das redes neurais, mapeamos a função $a(x_0)$ em uma dimensão de representação.

$$u(x) \quad \in \mathcal{R}^d \tag{29}$$
$$v(x) \quad \in \mathcal{R}^n \tag{30}$$

onde $n > d$

$$v_0(x) = NN_1(x, a(x))$$

$$v_{t+1}(x) = \sigma(W v_t(x) + \int_{B(x)} \|_\theta(x, y, a(x), a(y)) v_t(y) dy$$

$$u(x) = NN_2(v_T(x))$$

para $t = 1, \cdots, T$.
NN_1 e NN_2 são redes neurais feed-forward. (T não é o tempo e sim o numero de camadas)

- $\sigma => Relu$
- $W =>$ Transformação Linear

- Usando Gráfos.

$$\int \sim \sum$$

$$\int_{B(x)} \mathcal{K}_\theta(x, y, a(x), a(y)) v_t(y) dy \approx \frac{1}{|N|} \sum_{y \in N(x)} \mathcal{K}_\theta(x, y, a(x), a(y)) v_t(y)$$

onde $N(x)$ é o visinho de x de acordo com o grafo. $\mathcal{K}_\theta$ é um operador convolucional

Operador Neural de Fourier

Substitui o operador integral por um operador convolucional definido no espaço de Fourier

$$(\mathcal{F}f)_j(\kappa) == \int_D f_j(x)e^{-2i\pi(x,\kappa)}dx \tag{31}$$

$$(\mathcal{F}^{-1}f)_j(x) == \int_D f_j(\kappa)e^{-2i\pi(x,\kappa)}d\kappa \tag{32}$$

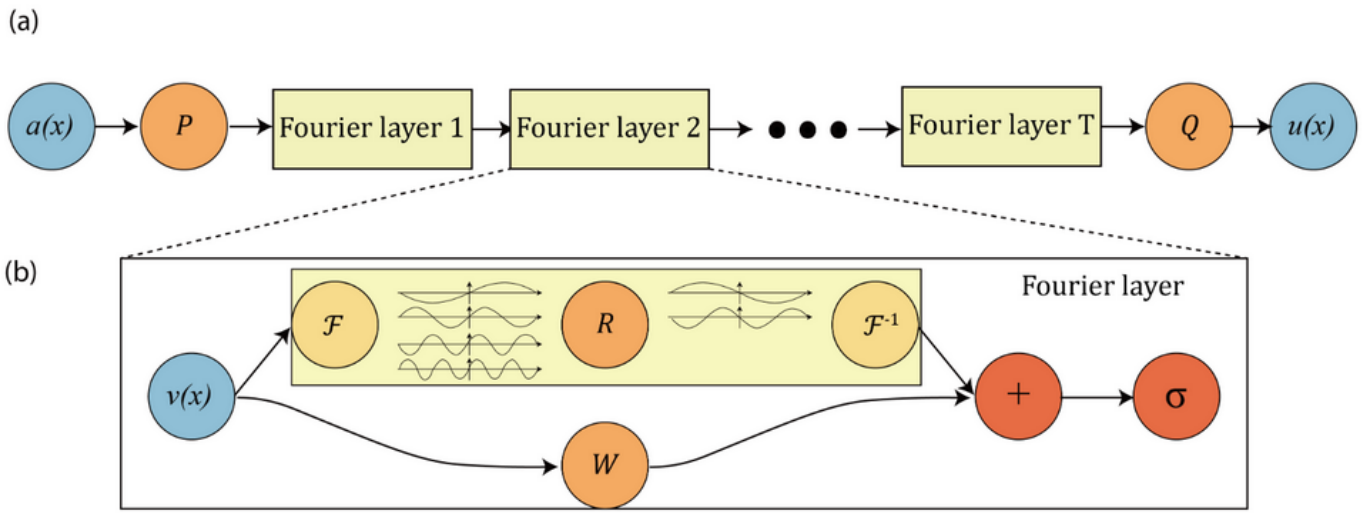
$$\mathcal{K}(a,\theta,v_t)(x) = \mathcal{F}^{-1}(\mathcal{F}(\mathcal{K}_\theta)\mathcal{F}(v_t))(x) \tag{33}$$

Onde podemos parametrizar no espaço de Fourier

$$\mathcal{K}(a, \theta, v_t)(x) = \mathcal{F}^{-1}(R_\theta(\mathcal{F}(v_t))(x)$$

Onde R_θ é a transformação de uma função periódica $\mathcal{K} : D- > \mathcal{R}^{v \times v}$ parametrizada por $\theta \in \Theta$. R_θ é parametrizado como um tensor complexo.
Não utilizamos todos os modos de Fourier, ou seja, truncamos a série de Fourier em um número máximo de modos ($\kappa_j \leq \kappa_{\max,j}$ para $(j = 1, \cdots, d)$. Note que assumimos que $\mathcal{K}$ é periódico e admite uma expanção de series de Fourier como modos discretos $\kappa \in \mathcal{Z}^d$.

Operador Neural de Fourier



Código:Operador Neural Fourier 1d