

# Mini-Curso: Introdução à Inteligência Artificial com Redes Neurais Artificiais



Parte 2

**Professor José Francisco Pessanha**

13 de dezembro de 2024

15:30 – 18:00, sala RAV62, 6º andar, bloco F

[https://homeprojextransicaoenergetica.netlify.app/\\_site/eventos](https://homeprojextransicaoenergetica.netlify.app/_site/eventos)

## Projeto de Extensão

**TRANSIÇÃO ENERGÉTICA: vantagens e desafios técnicos das energias renováveis para o equilíbrio entre custos, segurança e mudanças climáticas**



Departamento de  
Estatística



ELE  
Depto. de Eng. Elétrica

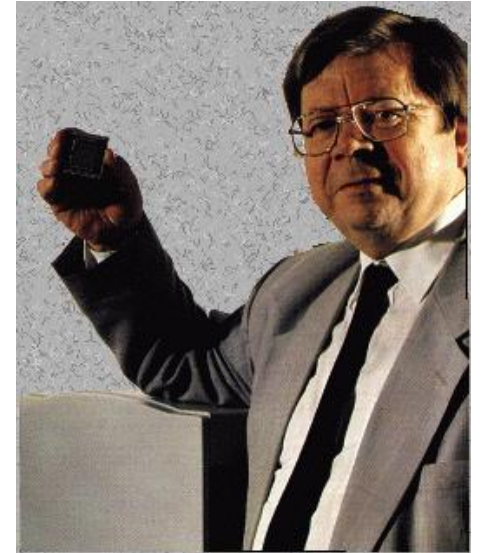


# Redes com treinamento não supervisionado

- Neste tipo de rede fornecemos dados não rotulados, ou seja, apenas os dados de entrada (sem dados de saída). O algoritmo sozinho identifica padrões e relacionamentos nos dados.
- Mapa Auto-Organizável (Self Organizing Map – SOM): aplicações em clustering e detecção de anomalias
- Autoencoder: detecção de anomalias, geração de dados sintéticos
- *Generative Adversarial Network* (GAN): geração de dados sintéticos

# Rede Auto-Organizável (Self-Organizing Map SOM)

- Proposto por Teuvo Kohonen em 1982
- Rede neural com treinamento não supervisionado
- A rede aprende as similaridades entre os padrões de entrada (reconhecimento de padrões)
- Útil na análise de agrupamentos (*cluster analysis*)



Teuvo Kohonen  
1934 – 2021

Biol. Cybern. 43, 59–69 (1982) [BF00337288.pdf](#)

---

Biological  
Cybernetics  
© Springer-Verlag 1982

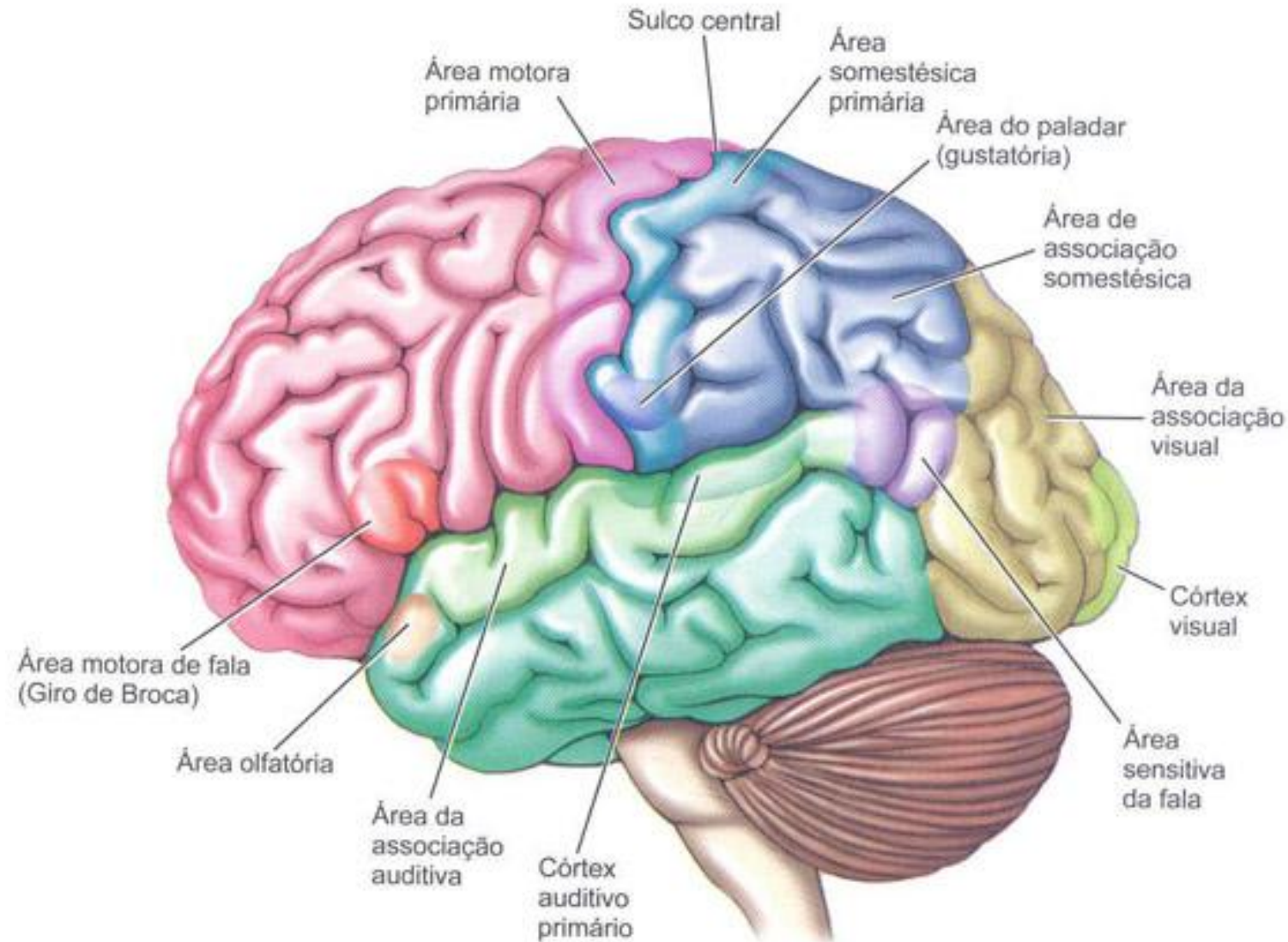
---

## Self-Organized Formation of Topologically Correct Feature Maps

Teuvo Kohonen

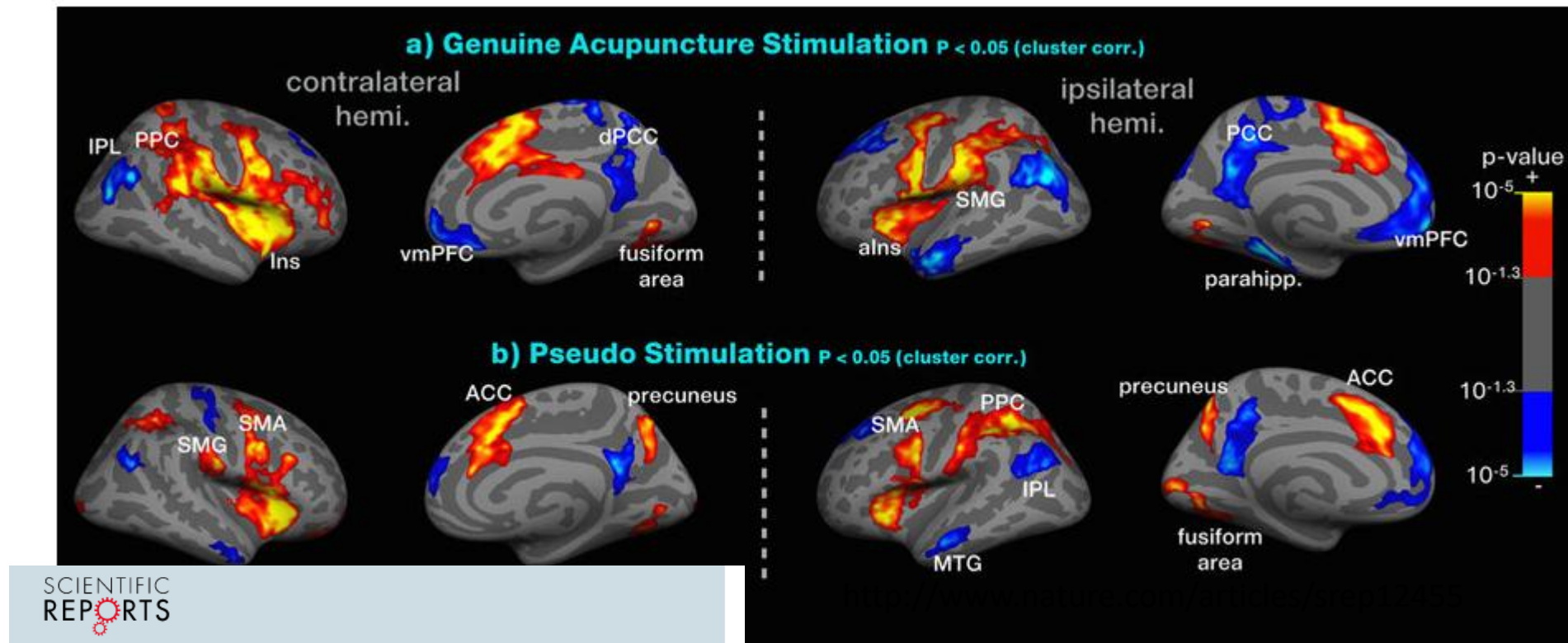
Department of Technical Physics, Helsinki University of Technology, Espoo, Finland

# Representação Neuro-Fisiológica







# Representação Neuro-Fisiológica



Altmetric: 5 Views: 1,092 [More detail >>](#)

Article | [OPEN](#)

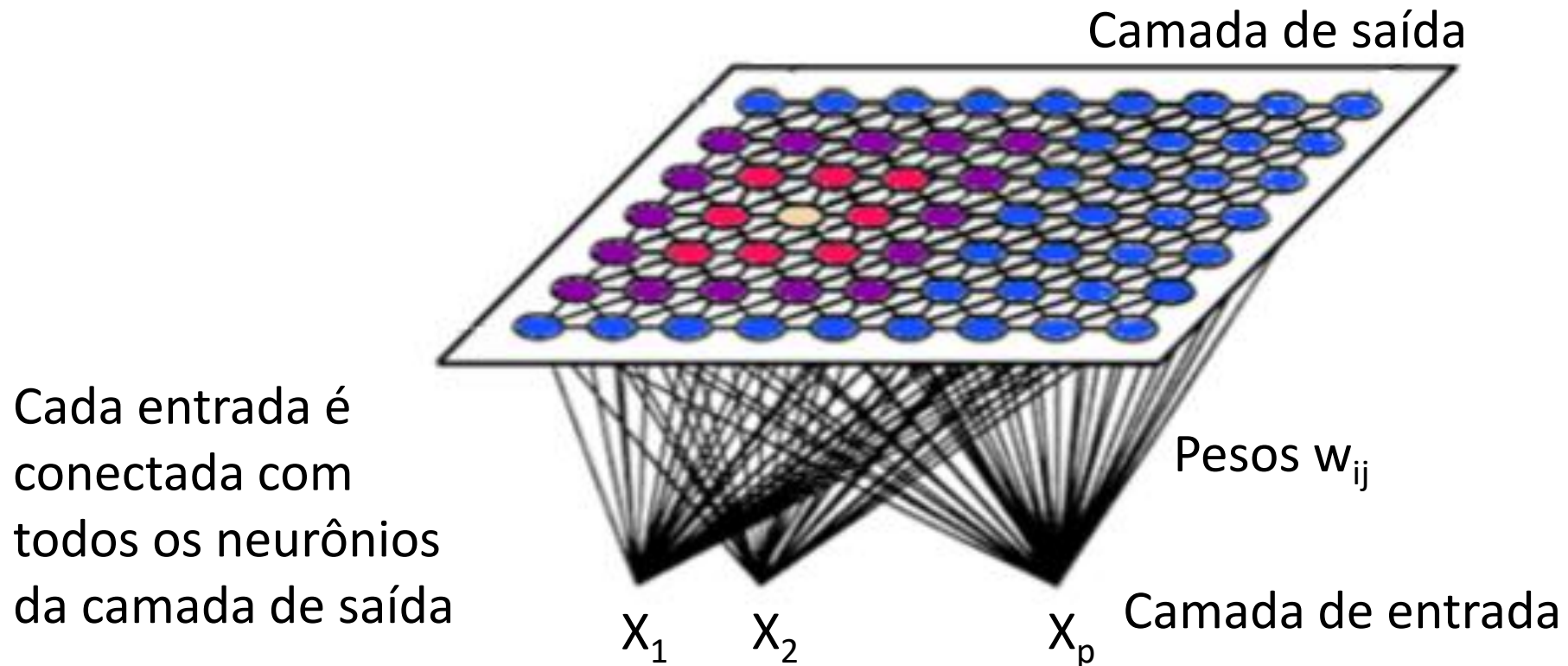
Cortical Activation Patterns of Bodily Attention triggered by Acupuncture Stimulation

Won-Mo Jung, In-Seon Lee , Christian Wallraven, Yeon-Hee Ryu, Hi-Joon Park & Younbyoung Chae 

Ativação cortical relacionada à  
acupuntura

# Arquitetura da rede SOM

- Em geral tem apenas duas camadas interconectadas por pesos sinápticos adaptáveis:
- Camada de entrada com  $p$  neurônios
- Camada de saída com  $q$  neurônios em uma grade bidimensional (mapa)



# Ajuste dos pesos (*codebook vectors*)

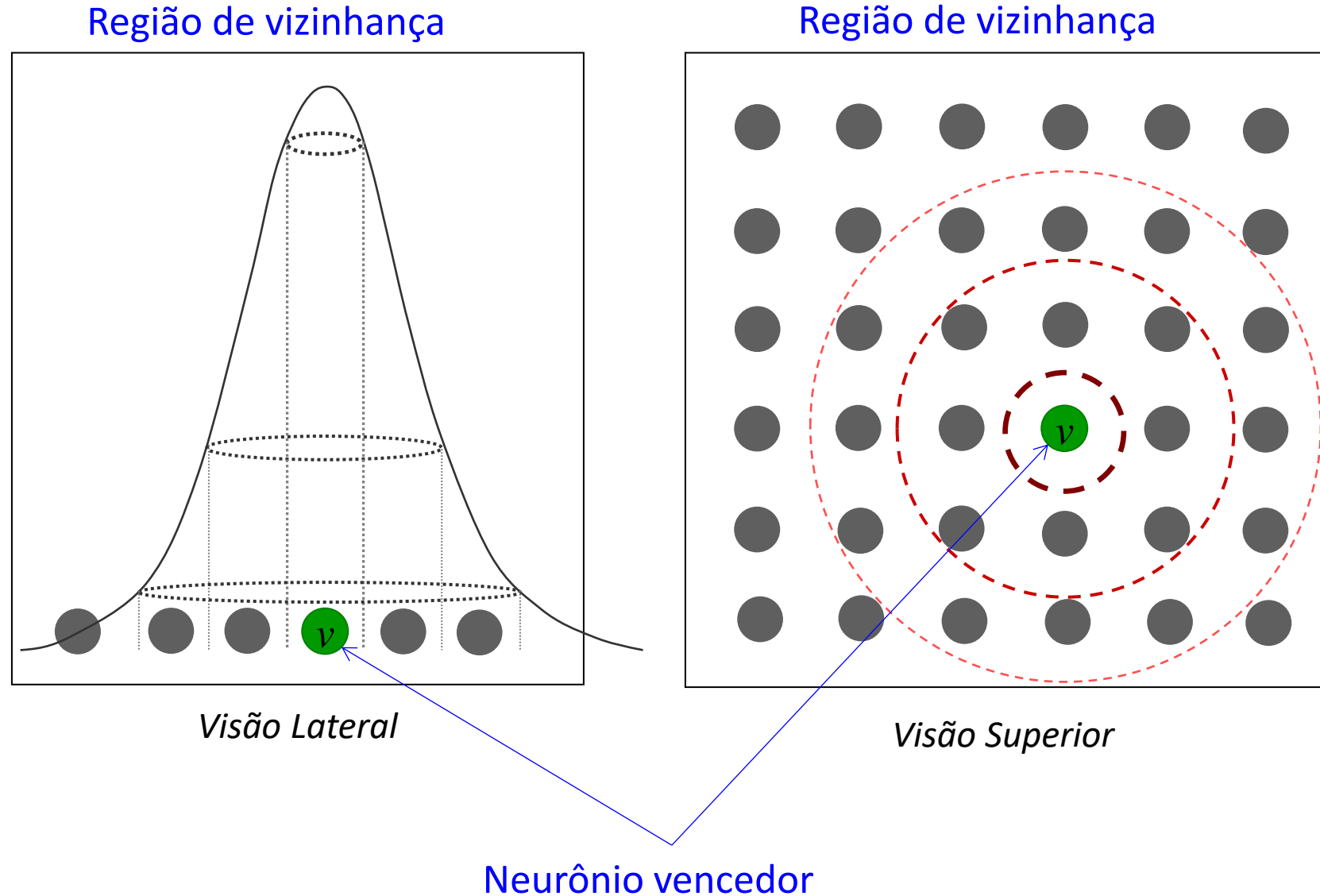
$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t)h_{vj}(t)[\mathbf{x} - \mathbf{w}_j(t)]$$

$\Delta \mathbf{w}$

Vetor peso atualizado      Vetor peso anterior      Taxa de aprendizagem      Vizinhança      Adaptação

- Durante o treinamento, os pesos são atualizados para se aproximarem dos dados de entrada com os quais estão mais próximos, seguindo um processo competitivo e cooperativo.
- Os pesos representam protótipos ou padrões presentes nos dados, possuem a mesma dimensão dos dados de entrada.

# Redução da região de vizinhança ao longo das iterações do treinamento





# Algoritmo de treinamento

- 1) Inicialize a taxa de aprendizado, o tamanho da região de vizinhança e faça a Inicialização aleatória dos pesos.
- 2) Para cada padrão de treinamento (vetor  $x_i \forall i=1,N$ ) faça
  - 2.1 Defina o neurônio vencedor
  - 2.2 Atualize os pesos do neurônio vencedor e dos vizinhos. Os pesos destes neurônios ficam mais próximos do padrão de entrada.
  - 2.3 Se o número de ciclos for múltiplo de N reduza a taxa de aprendizagem e a vizinhança do neurônio vencedor.
- 3) Repita o passo 2 até a convergência do mapa.

Duas fases: Ordenação e Convergência

**Ordenação:** Agrupa os neurônios em clusters (taxa de aprendizagem e região de vizinhança seguem trajetórias decrescentes)

**Convergência:** Refinamento do mapa topológica construído na ordenação (valores muito reduzidos da taxa de aprendizagem e vizinhança)

# Redução da região de vizinhança ao longo das iterações do treinamento

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t) h_{vj}(t) [\mathbf{x} - \mathbf{w}_j(t)]$$

Região de vizinhança gaussiana do neurônio vencedor

$$h_{vj}(t) = \exp\left(-\frac{\|\mathbf{r}_v - \mathbf{r}_j\|}{2\sigma^2(t)}\right)$$

distância entre o neurônio  $v$  vencedor e o neurônio  $j$  que está sendo atualizado

Define a largura da vizinhança e deve ser decrescente no tempo.

$$\sigma(t) = \sigma(0) \exp\left(-\frac{t}{\tau_1}\right) \Rightarrow h_{vj}(t) \rightarrow 0 \text{ quando } t \rightarrow \infty$$

$$\tau_1 = \frac{\text{Número\_de\_Iterações}}{\log \sigma(0)}$$

# Redução da região de vizinhança ao longo das iterações do treinamento

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \alpha(t) h_{vj}(t) [\mathbf{x} - \mathbf{w}_j(t)]$$

A taxa de aprendizagem decresce com o tempo, para que as adaptações sejam cada vez mais “finas”.

$$\alpha(t) = \alpha_0 \exp\left(-\frac{t}{\tau_2}\right),$$

número total de iterações

# Resultado do treinamento

- Ao final do treinamento os neurônios organizam-se em um mapa topologicamente ordenado, em que padrões semelhantes ativam neurônios vizinhos.
- Após o treinamento os neurônios especializaram-se em detectar características dos padrões de entrada.

# Rede SOM no R

## Package ‘kohonen’

September 4, 2015

**Version** 2.0.19

**Title** Supervised and Unsupervised Self-Organising Maps

**Author** Ron Wehrens

**Maintainer** Ron Wehrens <ron.wehrens@gmail.com>

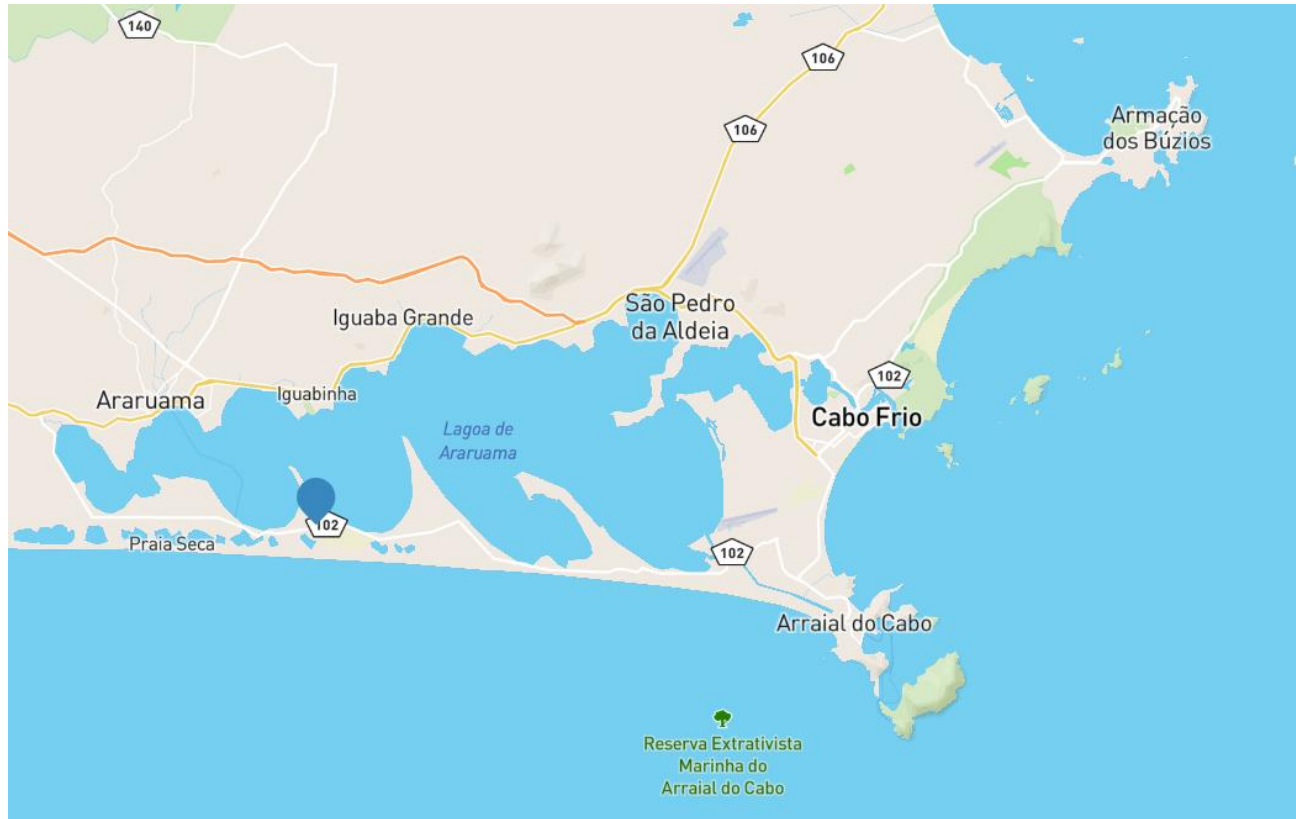
**Description** Functions to train supervised and self-organising maps (SOMs). Also interrogation of the maps and prediction using trained maps are supported. The name of the package refers to Teuvo Kohonen, the inventor of the SOM.



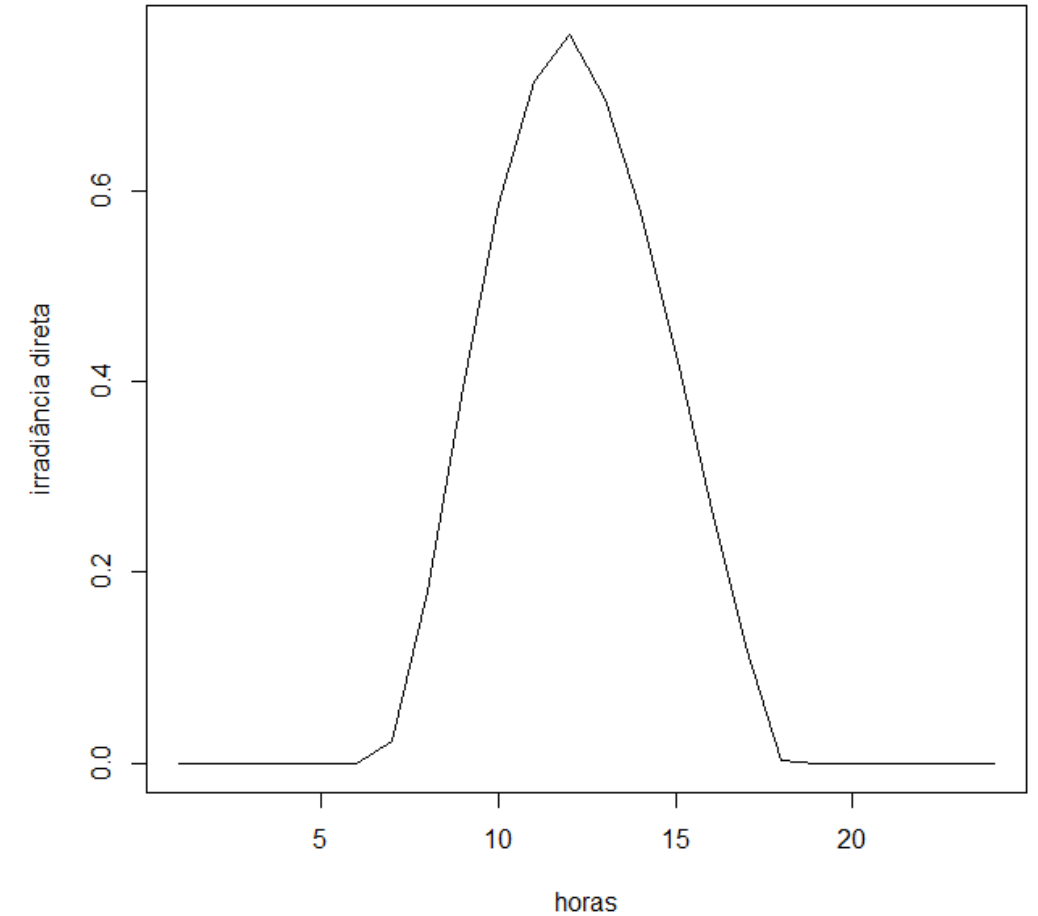


# Exemplo – Irradiância solar direta

Reanálises do MERRA 2 ([Renewables.ninja](https://renewables.ninja))



Irradiância direta ao longo do dia 1/1/2019  
(perfil da irradiância direta)



# Exemplo – Programa R

```
library(dplyr)
library(ggplot2)
library(kohonen)
```

## LEITURA DE DADOS

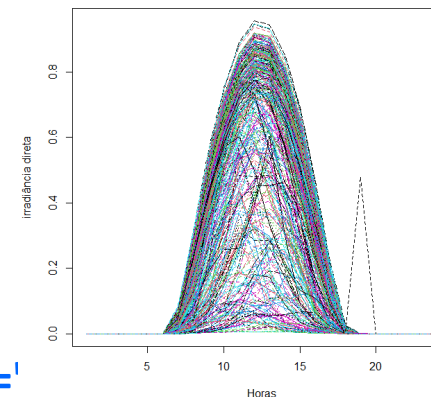
```
arquivo="c:/solar/dados_solar.csv"
dados=read.csv2(arquivo,header=T,dec=",")
attach(dados)
```

	A	B	C	D	E
1	local_time	electricity	irradiance_direct	irradiance_diffuse	temperature
2	01/01/2019 01:00	0	0	0	24,77
3	01/01/2019 02:00	0	0	0	24,647
4	01/01/2019 03:00	0	0	0	24,59
5	01/01/2019 04:00	0	0	0	24,554
6	01/01/2019 05:00	0	0	0	24,521
7	01/01/2019 06:00	1,497	0	0,03	24,71
8	01/01/2019 07:00	10,11	0,023	0,11	25,415
9	01/01/2019 08:00	25,762	0,18	0,135	26,524
10	01/01/2019 09:00	42,69	0,394	0,135	27,71
11	01/01/2019 10:00	56,422	0,587	0,135	28,737
12	01/01/2019 11:00	64,973	0,715	0,139	29,651
13	01/01/2019 12:00	68,276	0,764	0,146	30,332
14	01/01/2019 13:00	65,217	0,696	0,167	30,631
15	01/01/2019 14:00	58,469	0,579	0,181	30,621
16	01/01/2019 15:00	48,215	0,429	0,182	30,235
17	01/01/2019 16:00	35,322	0,274	0,165	29,702
18	01/01/2019 17:00	20,344	0,123	0,131	29,174
19	01/01/2019 18:00	6,067	0,002	0,084	28,336
20	01/01/2019 19:00	0,208	0	0,01	27,031
21	01/01/2019 20:00	0	0	0	26,294
22	01/01/2019 21:00	0	0	0	25,85
23	01/01/2019 22:00	0	0	0	25,567
24	01/01/2019 23:00	0	0	0	25,349

```
perfis=c()
for (dia in 1:364){
  inicio=(dia-1)*24+1
  fim=(dia-1)*24+24
  perfis=rbind(perfis,irradiance_direct[inicio:fim])
}
```

## MONTA PERFIS DA IRRADIÂNCIA DIRETA

```
windows()
matplot(matrix(seq(1,24,1),ncol=1),t(perfis),type='l',ylab='irradiância direta',xlab='
```

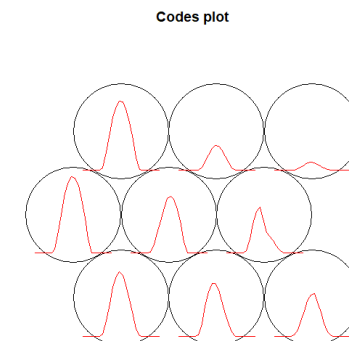


```
tamanho=3
dados=as.matrix(perfis)
set.seed(1513)
mapa=som(dados,grid=somgrid(tamanho,tamanho, "hexagonal"),rlen=100000)
```

## TREINA SOM

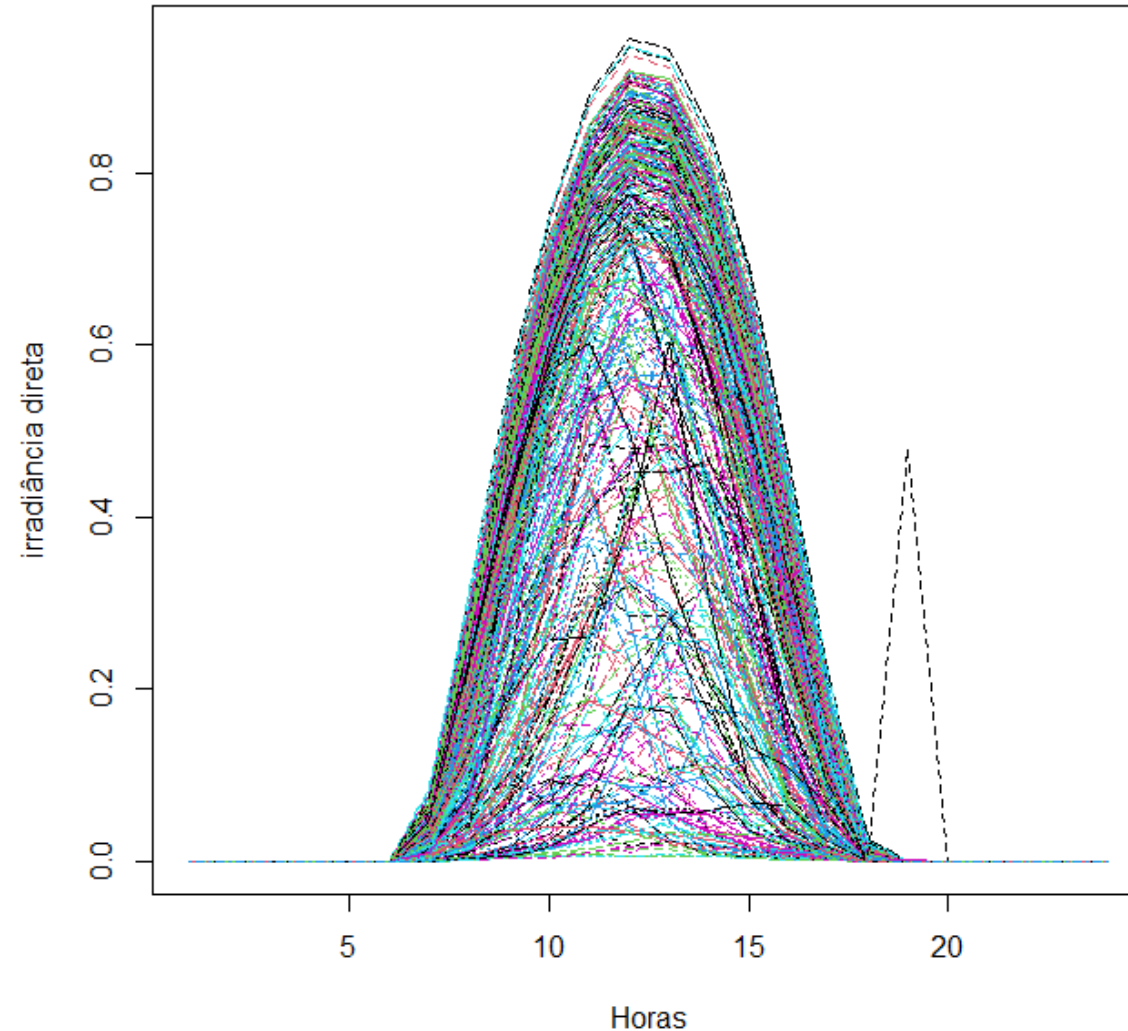
```
windows()
plot(mapa)
```

## MOSTRA OS VETORES DE PESOS DE CADA NEURÔNIO (CODE PLOT)



# Exemplo – Irradiância solar direta

- Perfis da irradiância direta ao longo dos dias do ano de 2019
- Cada curva é um perfil (vetor com 24 valores da irradiância direta horária)

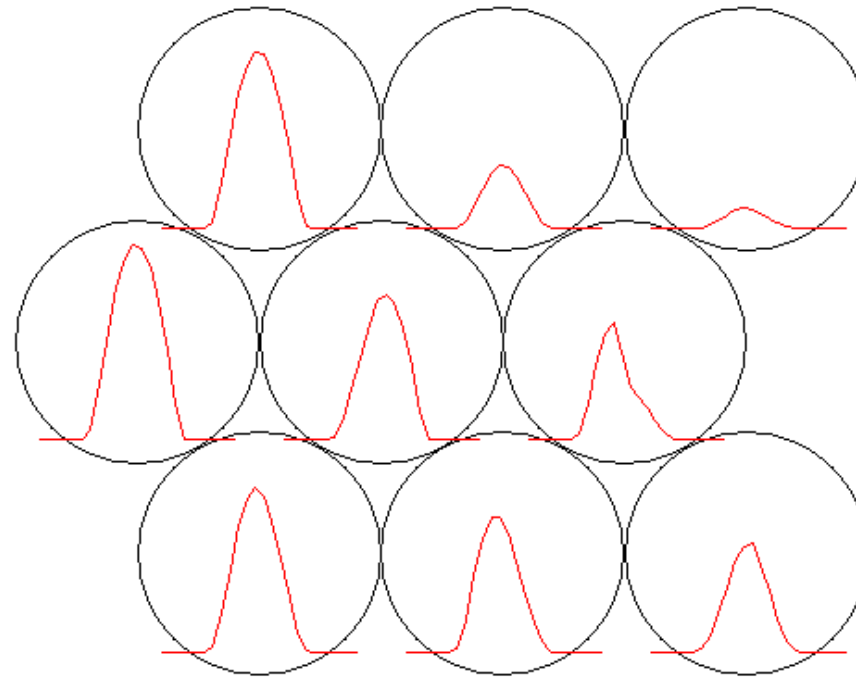


# Exemplo – Irradiância solar direta

Codes plot

Mapa 3 x 3 (9 neurônios)

Em cada neurônio, a curva representa o vetor de pesos que chegam ao neurônio



# Exemplo – Irradiância solar direta

**classe=mapa\$unit.classif**

```
[1] 1 1 8 6 8 1 7 5 4 7 1 7 7 2 1 6 7 7 7 8 3 2 7 7 9 3 7 5 7 8 4 4 4 7 9 3
[37] 9 9 7 4 4 6 3 9 7 9 9 6 8 8 7 9 5 3 1 3 9 1 9 9 9 8 8 1 8 3 4 1 8 9 9 9
[73] 8 3 3 9 8 9 5 8 2 7 4 4 4 7 2 7 4 4 7 4 7 4 7 1 4 1 8 5 7 4 5 9 5 2 5 4
[109] 4 4 4 4 1 7 1 4 4 4 4 5 7 1 1 1 4 7 5 1 7 7 4 4 7 3 9 8 8 9 9 7 4 4 5 5
[145] 1 1 4 4 5 7 7 3 6 1 5 3 1 7 2 7 7 1 7 1 7 4 4 7 4 7 7 1 2 7 7 4 7 5 7 4
[181] 4 4 1 7 1 9 5 1 5 5 7 4 4 4 4 7 3 3 2 3 1 1 4 4 4 4 4 4 4 1 1 4 4 4 9 9
[217] 8 8 7 4 4 4 4 4 9 8 4 4 4 2 9 2 1 8 8 1 7 5 9 3 7 4 5 9 7 1 9 8 7 4 4
[253] 4 4 4 7 3 5 4 4 4 1 1 2 8 7 5 9 9 1 1 3 5 4 4 4 4 4 4 7 2 9 9 8 4 4 4 4
[289] 7 7 4 1 7 8 9 1 7 1 2 7 1 8 1 3 3 5 4 6 1 8 8 9 9 9 8 5 3 9 9 3 1 8 9 9
[325] 3 9 9 5 5 7 6 3 9 3 9 2 3 5 3 9 7 7 1 9 9 9 9 9 7 7 7 2 9 9 9 9 9 7 7
[361] 7 7 7 5
```

**numel=table(classe) # número de perfis em cada neurônio**

**classe**

1	2	3	4	5	6	7	8	9
46	15	27	83	29	7	70	30	57

**tipo=mapa\$codes[[1]] # pesos que chegam aos neurônios**

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]	[,13]	[,14]	[,15]	[,16]	[,17]	[,18]	[,19]	[,20]	[,21]	[,22]	[,23]	[,24]
v1	0	0	0	0	0	0	0.024635847	0.16841333	0.37424625	0.56477176	0.6916381	0.73674089	0.69463523	0.58924727	0.41982176	0.23627975	0.077903696	0.0024719968	2.459840e-57	0	0	0	0	0
v2	0	0	0	0	0	0	0.017338344	0.14216280	0.34548962	0.50895607	0.6074115	0.60939427	0.53525969	0.40275008	0.25965920	0.13298584	0.044165773	0.0010223602	7.410985e-323	0	0	0	0	0
v3	0	0	0	0	0	0	0.010616469	0.07150984	0.17078476	0.29175123	0.4046197	0.47265759	0.48726479	0.39260718	0.27499647	0.14735243	0.044889783	0.0018034458	7.410985e-323	0	0	0	0	0
v4	0	0	0	0	0	0	0.042176711	0.25594031	0.49178938	0.68647938	0.8193370	0.88038867	0.86587491	0.77677822	0.61937859	0.40479212	0.163258928	0.0077030546	4.061403e-03	0	0	0	0	0
v5	0	0	0	0	0	0	0.015325547	0.09469892	0.23183087	0.37401201	0.5184697	0.62574439	0.65726480	0.62258189	0.50085800	0.32883910	0.129707047	0.0051766797	9.699665e-05	0	0	0	0	0
v6	0	0	0	0	0	0	0.027004961	0.16176084	0.33921798	0.47641573	0.5271956	0.39491943	0.23669437	0.19252365	0.14305149	0.08508013	0.033824837	0.0005336527	7.410985e-323	0	0	0	0	0
v7	0	0	0	0	0	0	0.030726628	0.20723932	0.42633481	0.61192577	0.7384029	0.79847166	0.78909670	0.70506336	0.55389481	0.35211904	0.141360662	0.0064045363	2.223295e-322	0	0	0	0	0
v8	0	0	0	0	0	0	0.005807462	0.03857016	0.09770298	0.17965373	0.2529725	0.28724183	0.27618486	0.23748442	0.16086592	0.08402044	0.027707978	0.0012909058	7.410985e-323	0	0	0	0	0
v9	0	0	0	0	0	0	0.002583698	0.01522882	0.03559886	0.06106917	0.0850042	0.09330849	0.08487752	0.07065632	0.04637701	0.02528758	0.008880193	0.0005501200	5.434722e-323	0	0	0	0	0



# Exemplo – Irradiância solar direta

**CALCULA PERFIL MÉDIO  
EM CADA NEURÔNIO**

```
classe=mapa$unit.classif
numel=table(classe) # número de perfis em cada neurônio
tipo=mapa$codes[[1]] # pesos que chegam aos neurônios
```

```
windows()
par(mfrow=c(tamanho,tamanho))
par(mar=c(1,1,1,1))
indices = apply(matrix(seq(tamanho^2:1),tamanho,tamanho),1,rev)
vecindices = matrix(t(indices),1,tamanho^2)
```

```
for (i in 1:tamanho^2) {
```

```
  aux = which(classe==vecindices[i])
```

```
  teto = max(dados[aux,])
```

```
  piso = min(dados[aux,])
```

```
  ncurvas = length(aux)
```

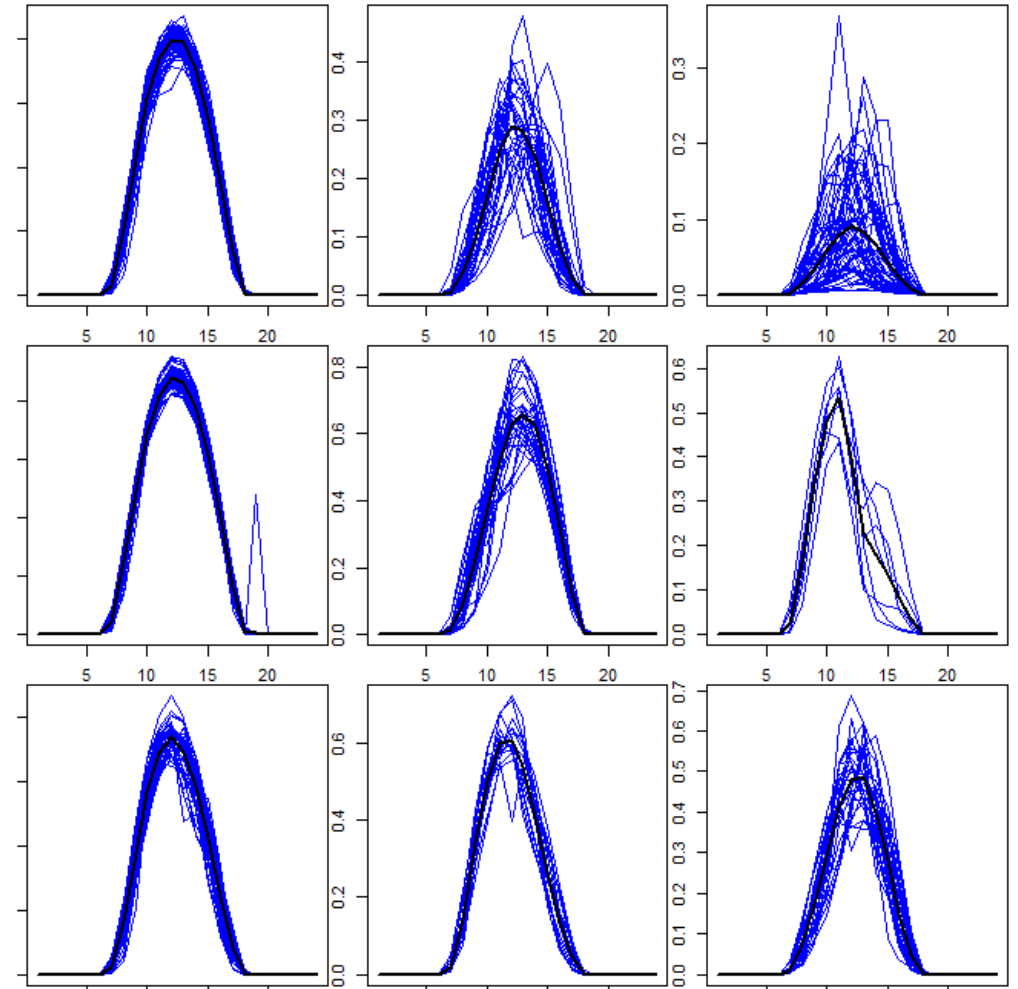
```
  if (ncurvas>0) {
```

```
    plot(tipo[vecindices[i],],type = "l",ylim=c(piso,teto),ylab = "", xlab = "",lty = "solid",col="black",lwd=2)
```

```
    for (j in 1:ncurvas) { lines(dados[aux[j],],col = "blue", lty = 1,ylab = "", xlab = "") }
```

```
    lines(tipo[vecindices[i],],type = "l",main = paste("cluster ",vecindices[i]),ylab = "", xlab = "",lty = "solid",col="black",lwd=2) }}
```

**FAZ GRÁFICO DOS PERFIS  
EM CADA NEURÔNIO**



# Exemplo – Irradiância solar direta

**CALCULA PERFIL MÉDIO  
EM CADA NEURÔNIO**

```
classe=mapa$unit.classif
numel=table(classe) # número de perfis em cada neurônio
tipo=mapa$codes[[1]] # pesos que chegam aos neurônios
```

```
windows()
par(mfrow=c(tamanho,tamanho))
par(mar=c(1,1,1,1))
indices = apply(matrix(seq(tamanho^2:1),tamanho,tamanho),1,rev)
vecindices = matrix(t(indices),1,tamanho^2)
```

```
for (i in 1:tamanho^2) {
```

```
  aux = which(classe==vecindices[i])
```

```
  teto = max(dados[aux,])
```

```
  piso = min(dados[aux,])
```

```
  ncurvas = length(aux)
```

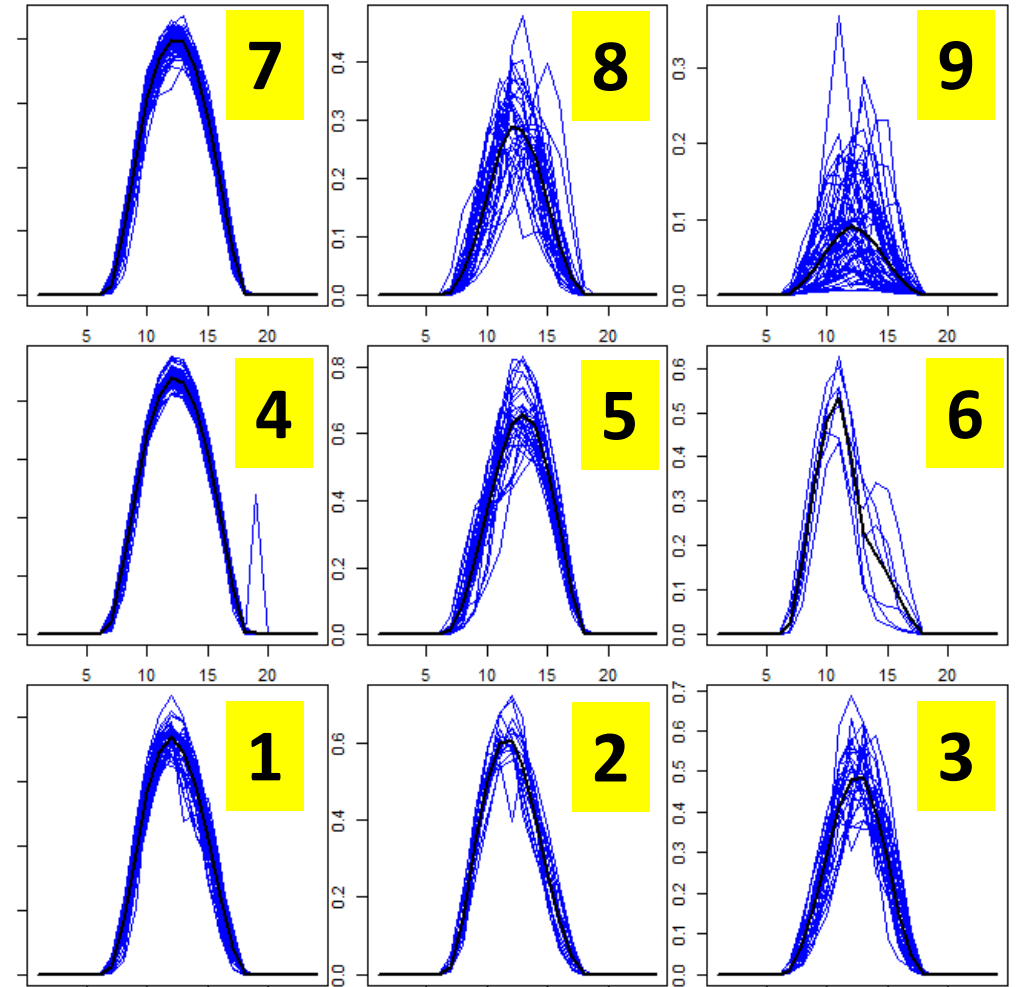
```
  if (ncurvas>0) {
```

```
    plot(tipo[vecindices[i],],type = "l",ylim=c(piso,teto),ylab = "", xlab = "",lty = "solid",col="black",lwd=2)
```

```
    for (j in 1:ncurvas) { lines(dados[aux[j],],col = "blue", lty = 1,ylab = "", xlab = "") }
```

```
    lines(tipo[vecindices[i],],type = "l",main = paste("cluster ",vecindices[i]),ylab = "", xlab = "",lty = "solid",col="black",lwd=2) }}
```

**FAZ GRÁFICO DOS PERFIS  
EM CADA NEURÔNIO**



# Exemplo – Irradiância solar direta

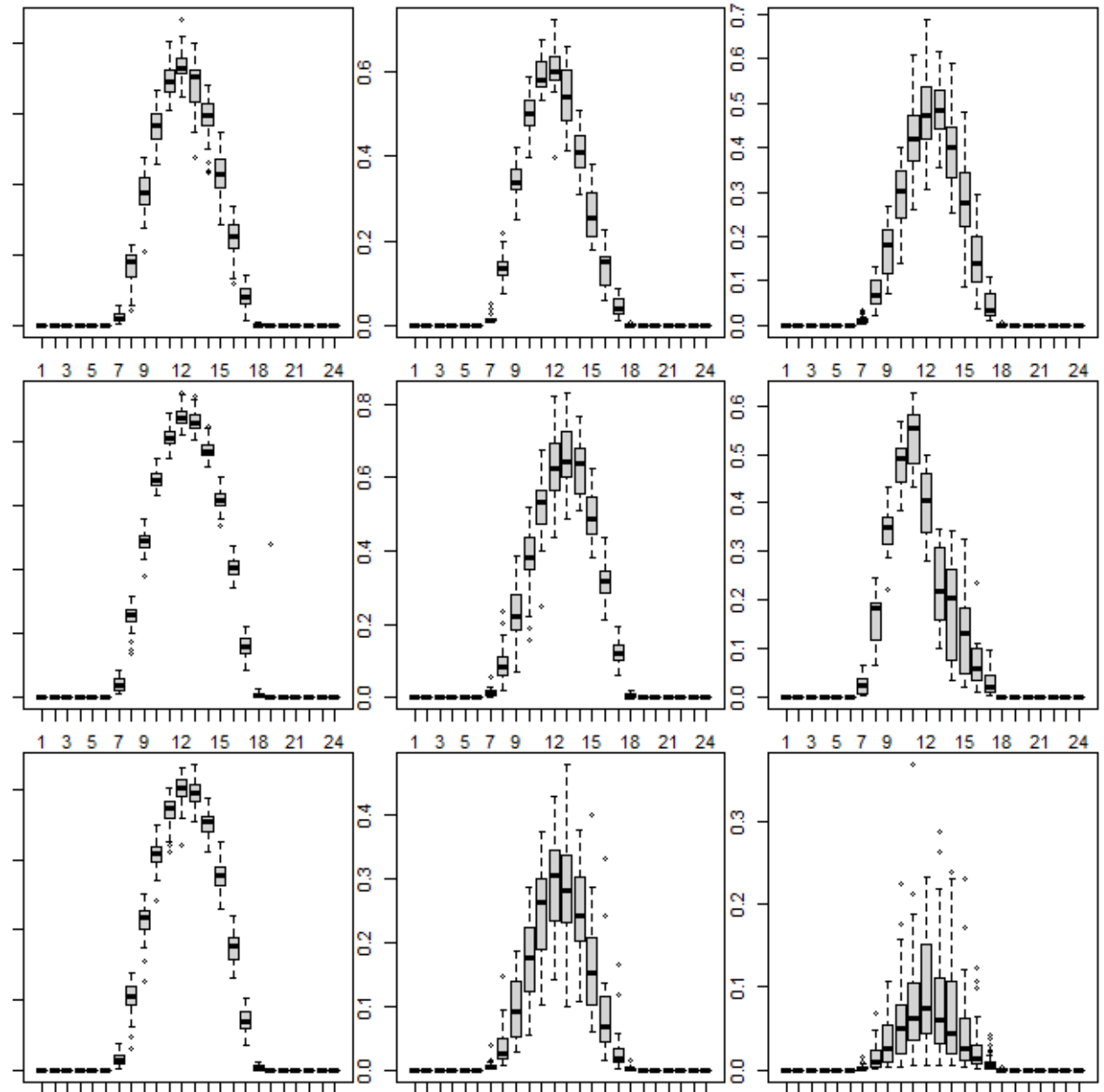
IEEE LATIN AMERICA TRANSACTIONS, VOL. 18, NO. 9, SEPTEMBER 2020

## An Approach for Data Treatment of Solar Photovoltaic Generation

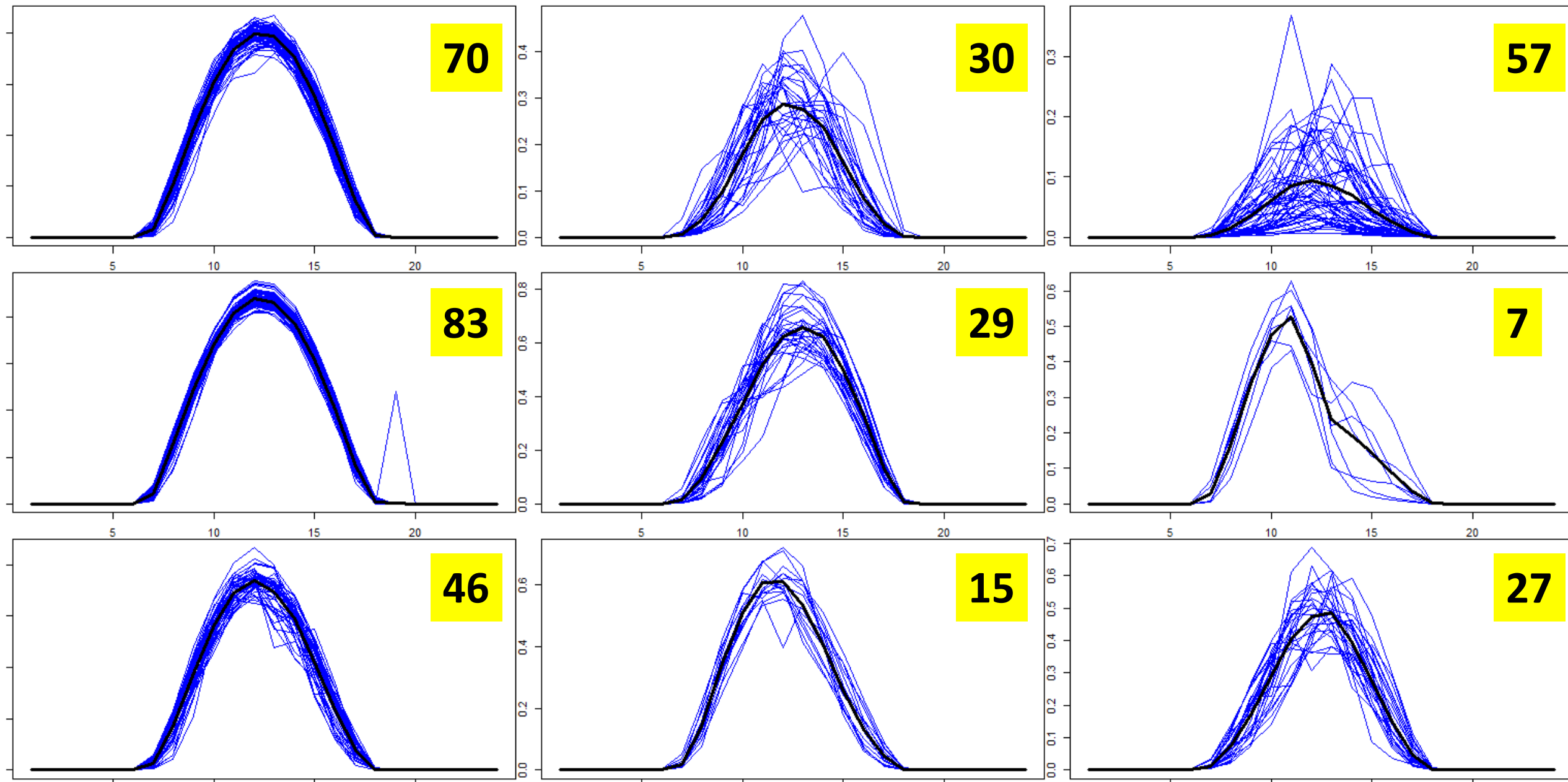
J. F. M. Pessanha, A. C. G. Melo, R. P. Caldas and D. M. Falcão

Boxplots dos perfis  
em cada neurônio

```
windows()  
par(mfrow=c(tamanho,tamanho))  
par(mar=c(1,1,1,1))  
for (i in 1:(tamanho*tamanho)) {  
  indice=which(classe==i)  
  numel=c(numel,length(indice))  
  boxplot(dados[indice,])  
}
```

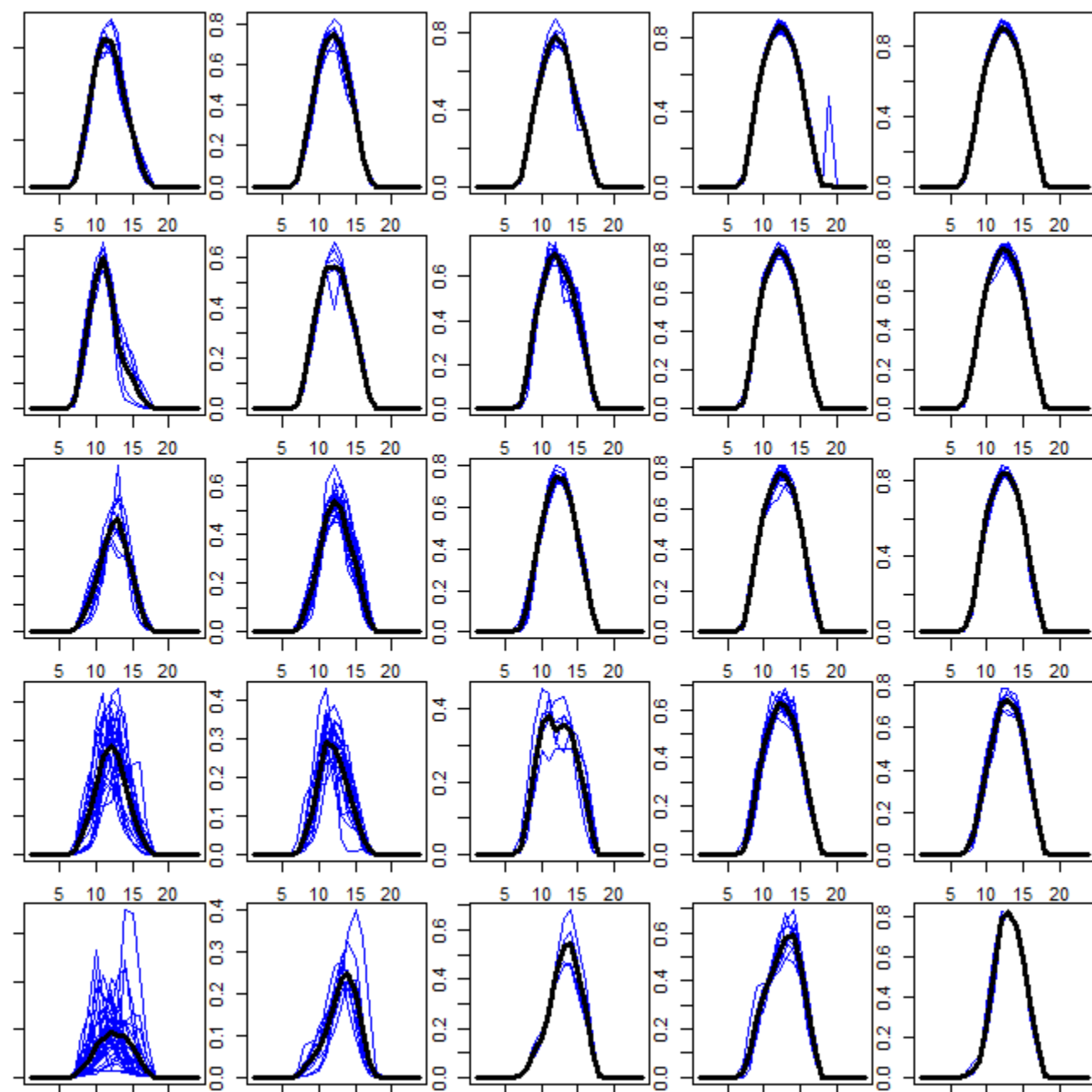
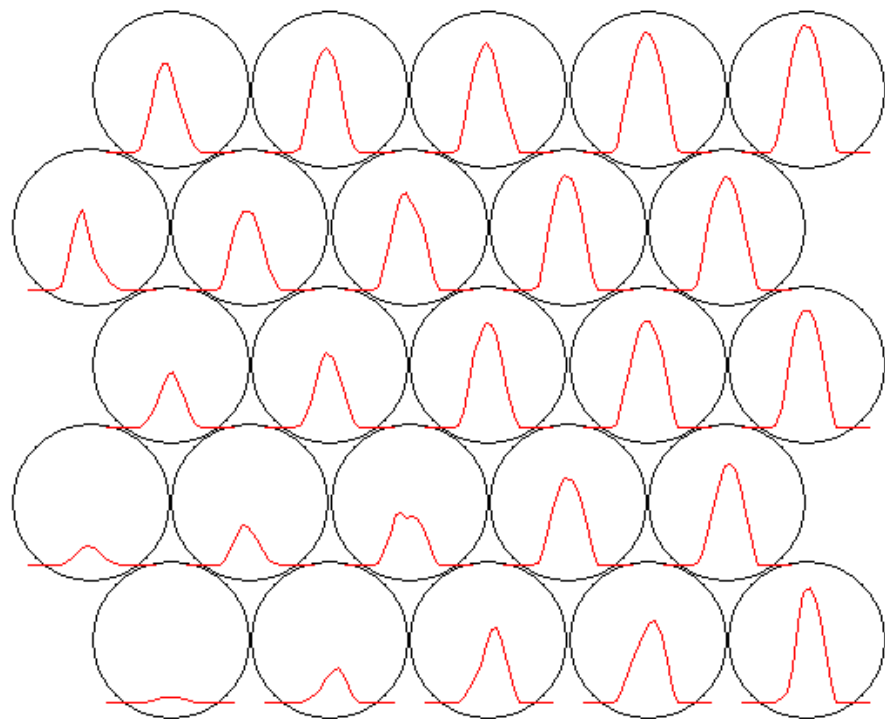


# Número de curvas em cada neurônio (*cluster*)



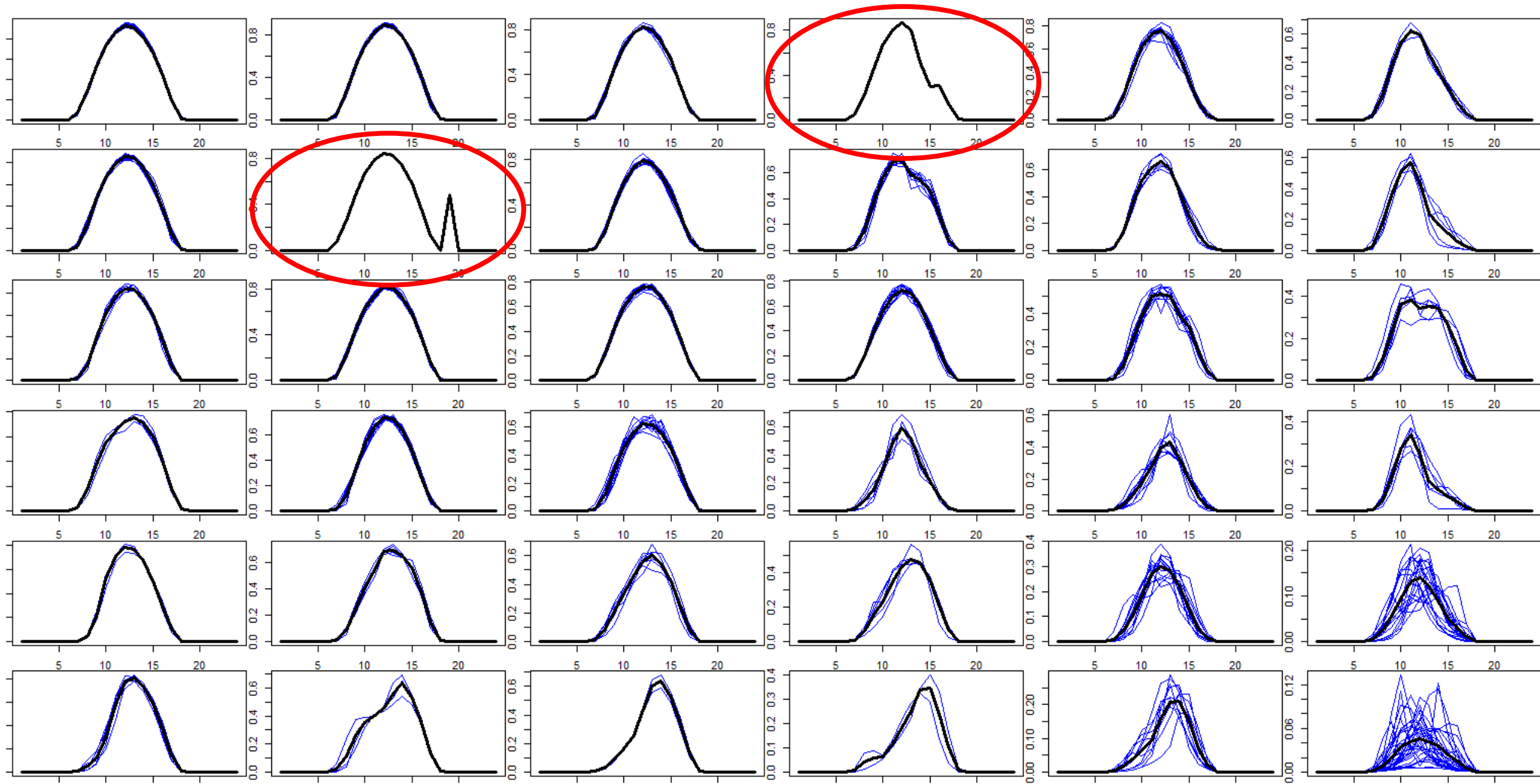
# SOM 5x5

Codes plot





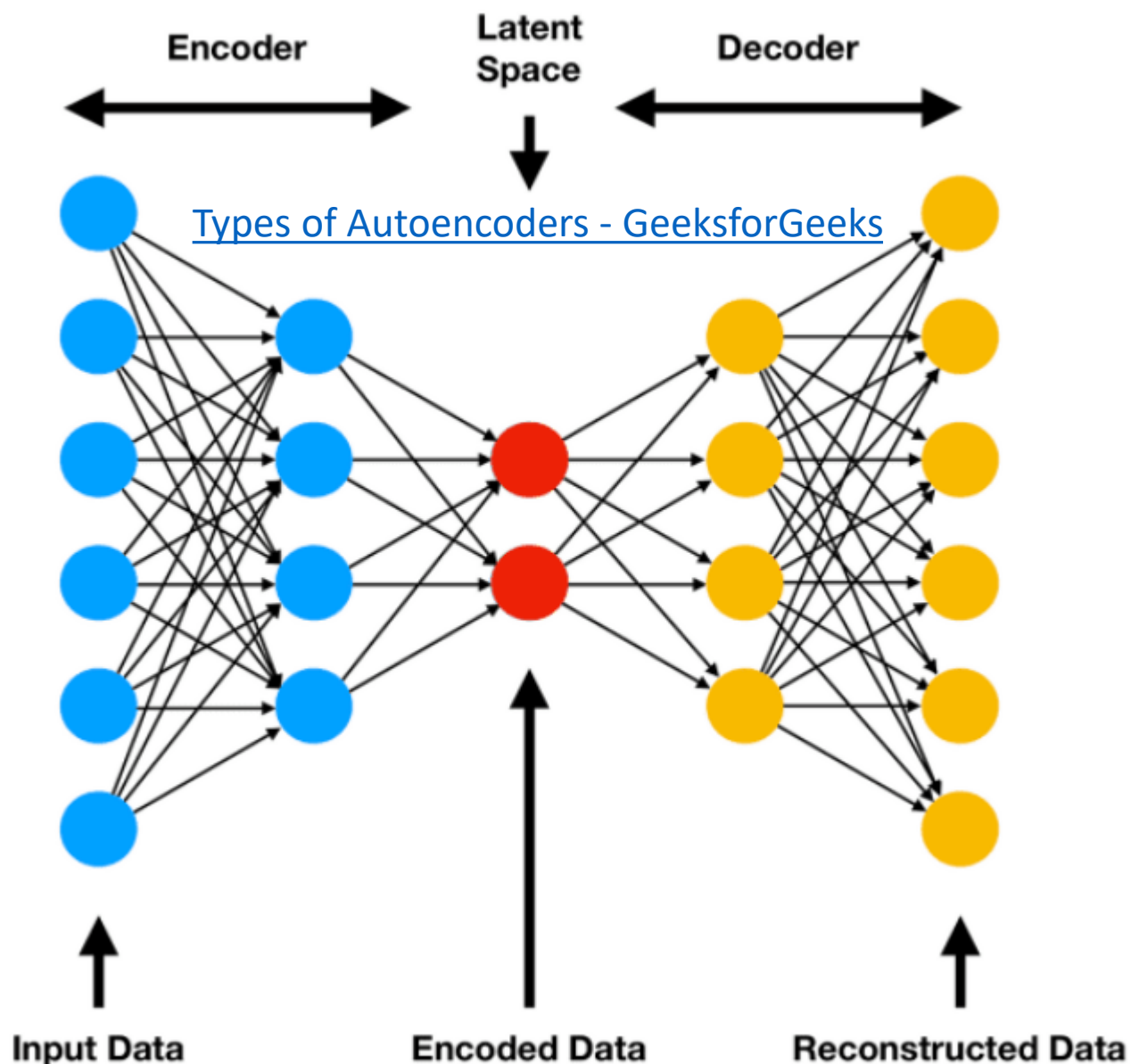
# SOM 6x6



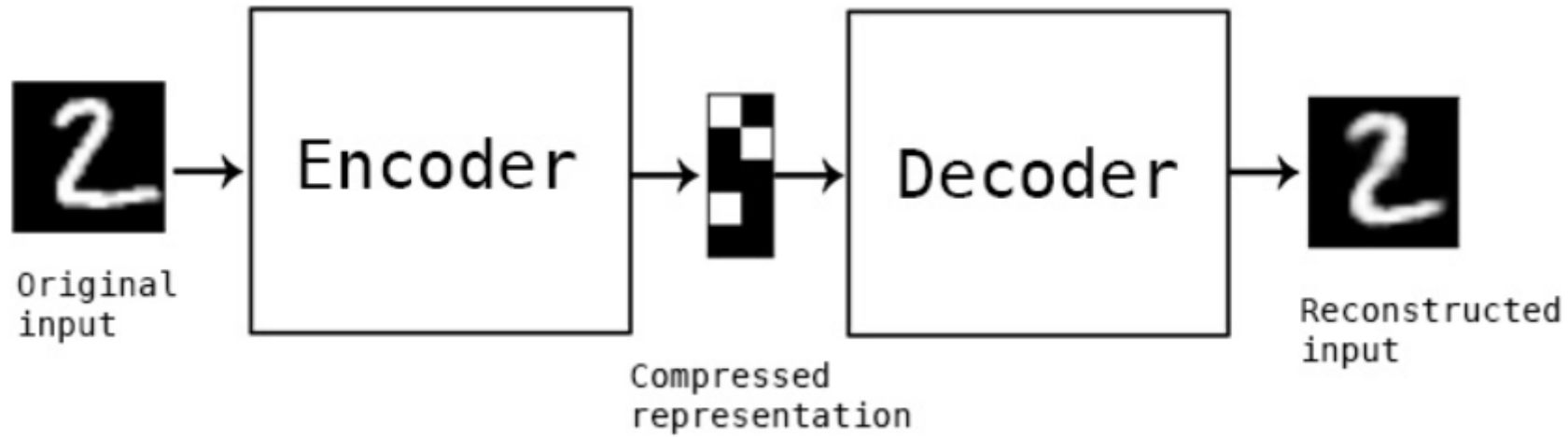
# Autoencoder

Um **autoencoder** é um tipo de rede neural projetada para aprender uma representação compacta e eficiente (codificação) dos dados, geralmente com o objetivo de compressão, redução de dimensionalidade ou aprendizado de características relevantes.

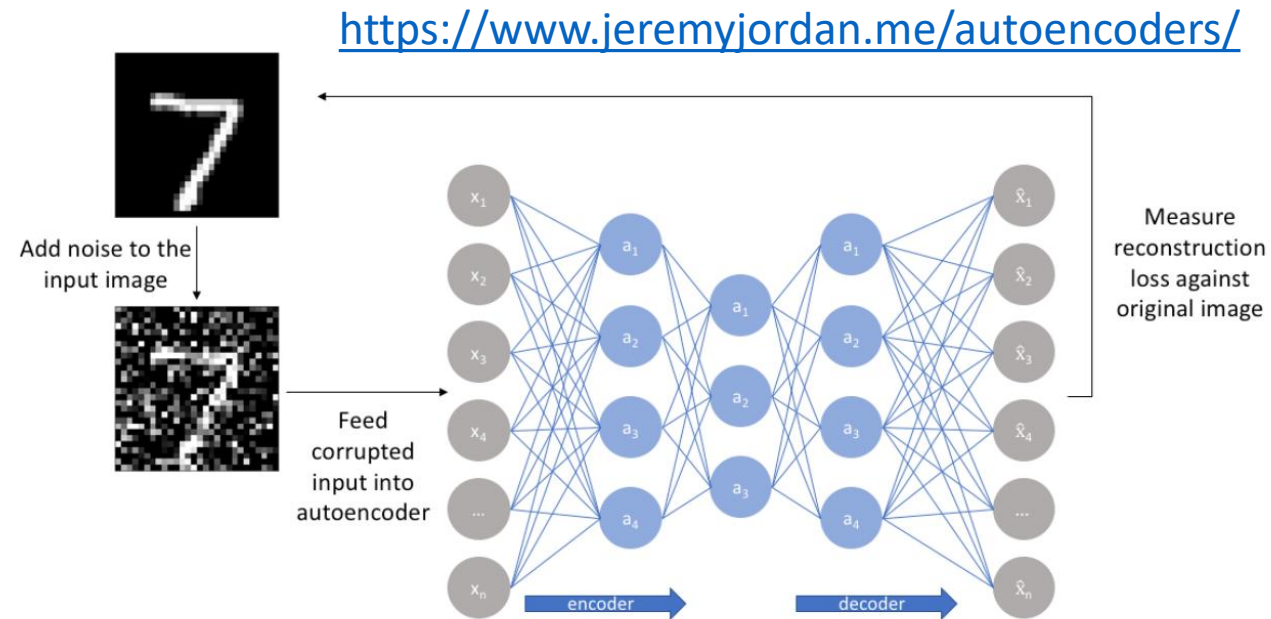
Ele é composto de duas partes principais: o **encoder** (codificador) e o **decoder** (decodificador).



# Autoencoder

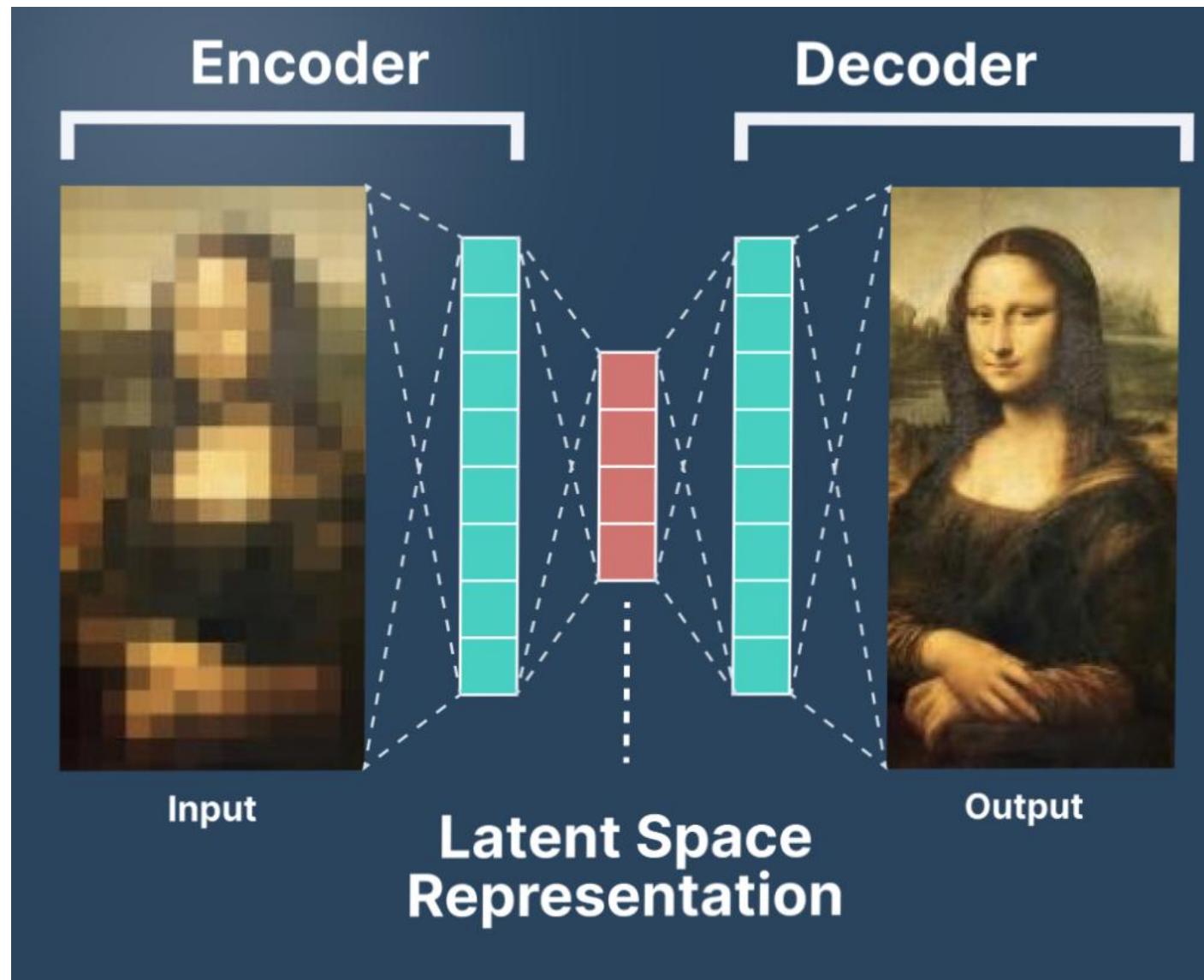


<https://blog.curso-r.com/posts/2017-06-26-construindo-autoencoders/>



# Autoencoder

<https://learnius.com/slp/9+Speech+Synthesis/1+Fundamental+Concepts/8+Speaker+and+Style+Embeddings/autoencoder>



# Estrutura do Autoencoder

## 1. Encoder (Codificador):

- Transforma os dados de entrada em uma representação de dimensionalidade reduzida, chamada de **código** ou **latente**.
- Sua função é aprender a mapear os dados de entrada ( $x$ ) para um espaço latente ( $z$ ).

$$z = f(x)$$

## 2. Latent Space (Espaço Latente):

- Contém a codificação comprimida dos dados, que deve capturar as informações mais importantes.

## 3. Decoder (Decodificador):

- Reconstrói os dados originais a partir da representação latente  $z$ .
- Sua função é aprender a mapear  $z$  de volta para  $x'$ , uma aproximação dos dados de entrada.

$$x' = g(z)$$

## Perda (Loss):

- O objetivo do treinamento é minimizar a diferença entre os dados de entrada ( $x$ ) e os dados reconstruídos ( $x'$ ).
- A métrica comumente usada é o **Erro Quadrático Médio (MSE)** ou outra medida de similaridade.

$$\text{Loss} = ||x - x'||^2$$



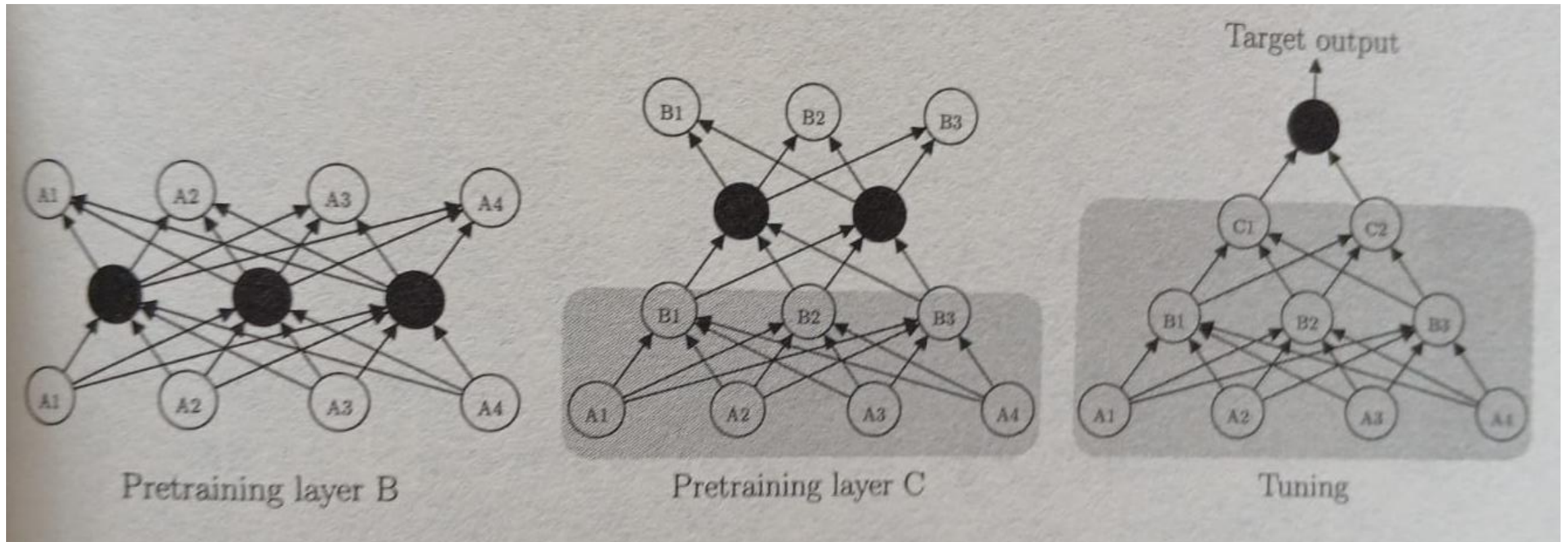
# Aplicação do Autoencoder

- 1.Redução de dimensionalidade:** Similar ao PCA (Análise de Componentes Principais), porém mais flexível, já que pode modelar relações não lineares nos dados.
- 2.Detecção de anomalias:** Como os autoencoders aprendem padrões comuns nos dados, podem ser usados para identificar exemplos que não conseguem ser reconstruídos bem (potenciais anomalias).
- 3.Compressão de dados:** Usado para comprimir dados em um formato mais eficiente, preservando informações importantes.
- 4.Pre-treinamento:** Pode ser usado para inicializar redes profundas com boas representações dos dados.
- 5.Geração de dados:** Autoencoders variantes, como o **Variational Autoencoder (VAE)**, são usados para gerar dados novos com características semelhantes aos dados originais.

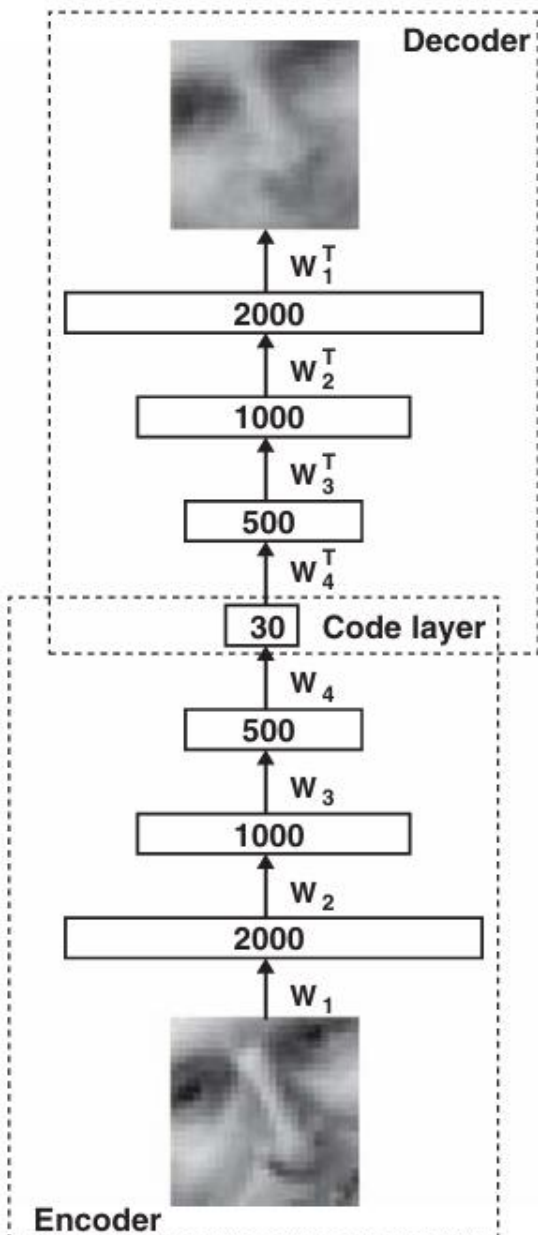
# Greedy layer-wise pretraining (Kelleher, 2019)

Abordagem **hierárquica e incremental** para treinar redes neurais profundas camada por camada, em vez de treinar toda a rede simultaneamente. A ideia é treinar uma camada de cada vez, garantindo que cada uma aprenda boas representações antes de passar para a próxima. **O termo "greedy" refere-se ao fato de que cada camada é treinada de forma independente, sem considerar as camadas futuras.**

Técnica de treinamento utilizada principalmente para redes neurais profundas com o objetivo de superar dificuldades associadas ao treinamento direto (end-to-end) de redes muito profundas. Essa abordagem foi especialmente popular antes do avanço de técnicas como inicializações mais robustas de pesos e otimizadores modernos (por exemplo, Adam).



# Deep Autoencoder Networks



Science

Current Issue

First release papers

Archive

About ▾

Submit manuscript

HOME > SCIENCE > VOL. 313, NO. 5786 > REDUCING THE DIMENSIONALITY OF DATA WITH NEURAL NETWORKS

REPORTS

f X in 5 6 7 8

## Reducing the Dimensionality of Data with Neural Networks

G. E. HINTON AND R. R. SALAKHUTDINOV [Authors Info & Affiliations](#)

SCIENCE • 28 Jul 2006 • Vol 313, Issue 5786 • pp. 504-507 • DOI: 10.1126/science.1127647

38,713 13,947



CHECK ACCESS

## Reducing the Dimensionality of Data with Neural Networks

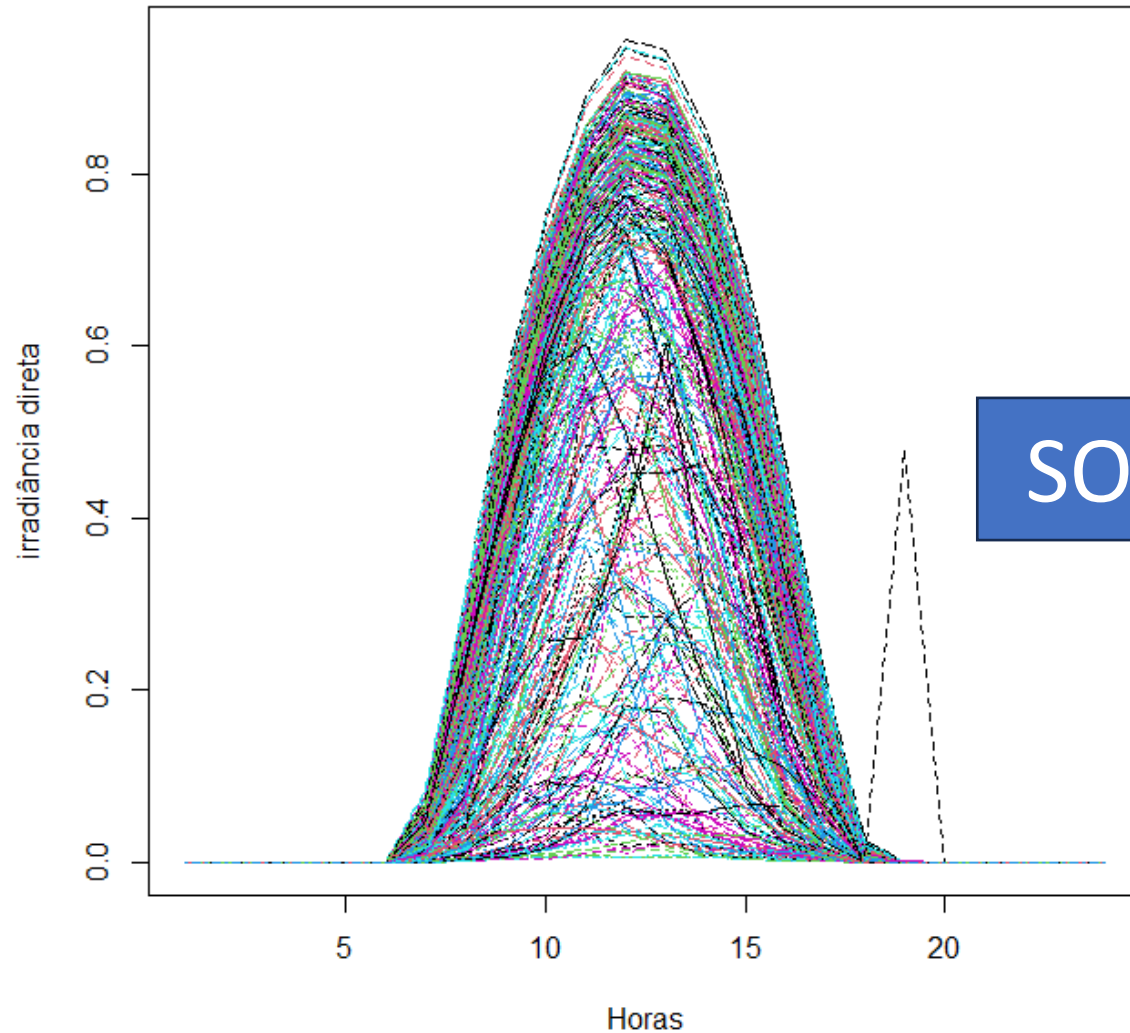
G. E. Hinton\* and R. R. Salakhutdinov

High-dimensional data can be converted to low-dimensional codes by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Gradient descent can be used for fine-tuning the weights in such “autoencoder” networks, but this works well only if the initial weights are close to a good solution. We describe an effective way of initializing the weights that allows deep autoencoder networks to learn low-dimensional codes that work much better than principal components analysis as a tool to reduce the dimensionality of data.

<https://www.science.org/doi/abs/10.1126/science.1127647>

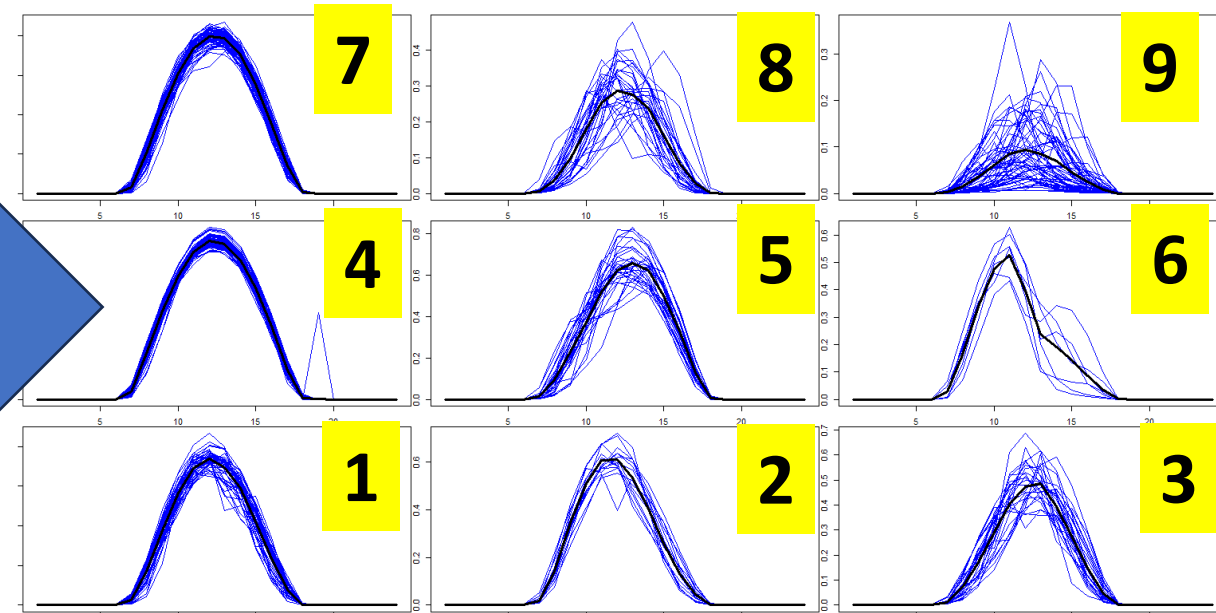
# Exemplo – Irradiância solar direta

- Perfis da irradiância direta ao longo dos dias do ano de 2019
- Cada curva é um perfil (vetor com 24 valores da irradiância direta horária)



SOM

9 Clusters



**Vamos treinar um autoencoder para cada cluster ou um conjunto de clusters**

# Autoencoder com o pacote ANN2 do R

```
library(ANN2)
```

```
indice=which(classe==4 | classe==7) # identifica perfis classificados nos clusters 4 e 7
```

```
aeNN <- autoencoder(perfis[indice,], hidden.layers=c(24,12,2,12,24), activ.functions="sigmoid", batch.size=1, optim.type="adam")  
plot(aeNN)
```

Arquitetura do  
autoencoder

Artificial Neural Network:

- Layer - 24 nodes - input
- Layer - 24 nodes - sigmoid
- Layer - 12 nodes - sigmoid
- Layer - 2 nodes - sigmoid
- Layer - 12 nodes - sigmoid
- Layer - 24 nodes - sigmoid
- Layer - 24 nodes - linear

with squared loss and Adam optimizer

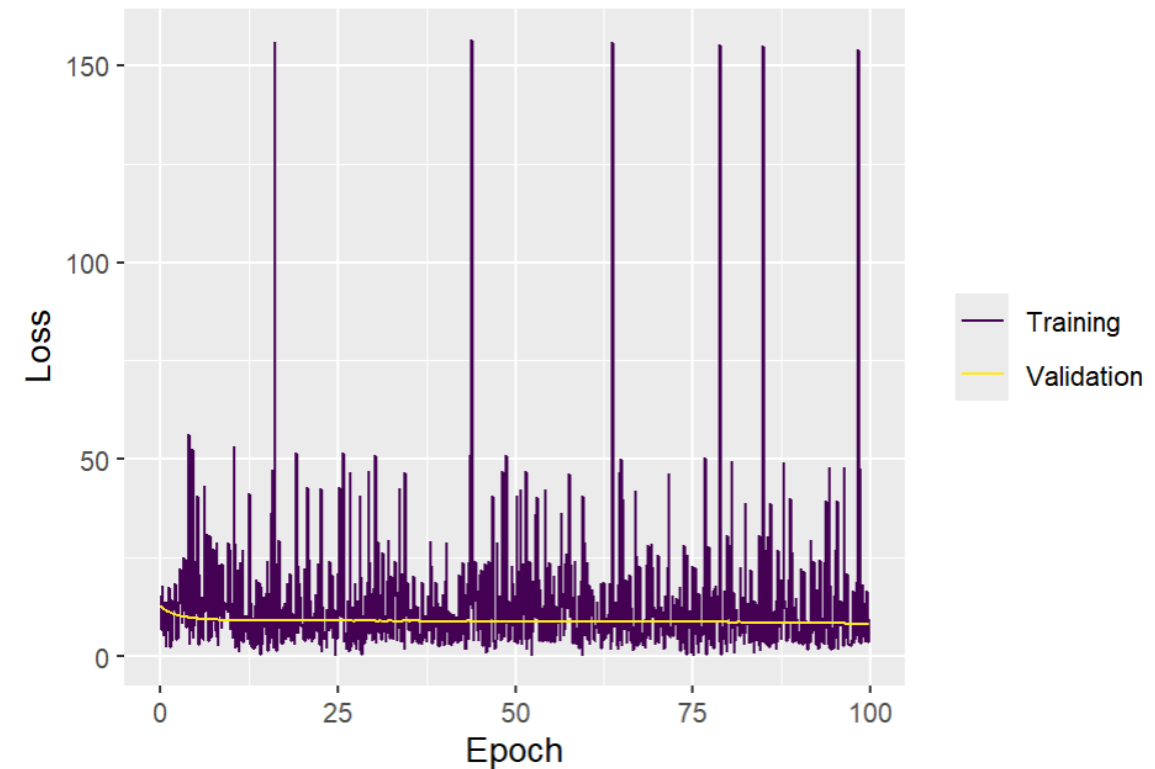
Training progress:

[+++++] 100% -

Validation loss: 8.25805

Mensagem de aviso:

small validation set, only 15 observations



10% dos dados são  
usados na validação



# Autoencoder com o pacote ANN2 do R

## # GERA UMA ENTRADA RUIDOSA

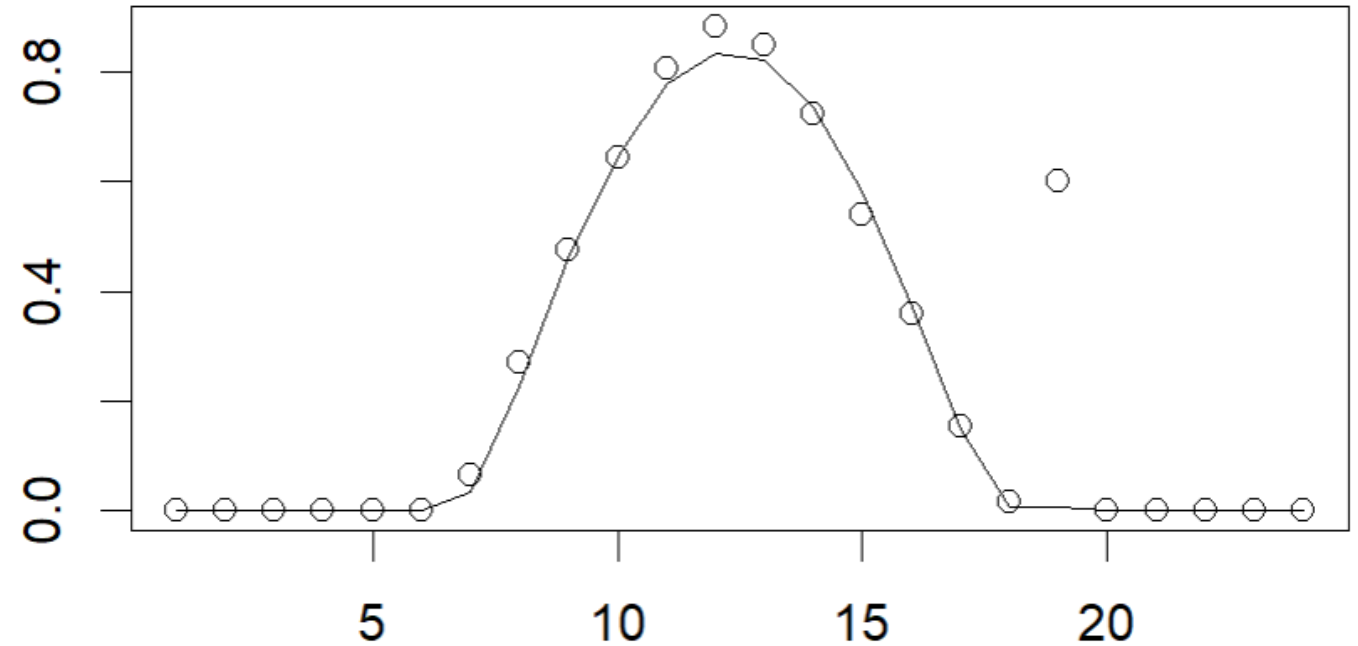
```
i=30  
aux=perfis[indice[i],]  
aux[19]=aux[19]+0.6
```

## # AVALIA A ENTRADA RUIDOSA

```
saida=predict(aeNN,newdata=t(aux))  
maximo=max(c(aux,as.numeric(saida$prediction)))
```

## # GRÁFICO DA ENTRADA RUIDOSA E DA SAÍDA GERADA PELO AUTOENCODER

```
plot(aux,ylim=c(0,maximo)) # entrada ruidosa  
lines(as.numeric(saida$prediction)) # resultado do autoencoder
```



# Autoencoder com o pacote ANN2 do R

## # GERA UMA ENTRADA RUIDOSA

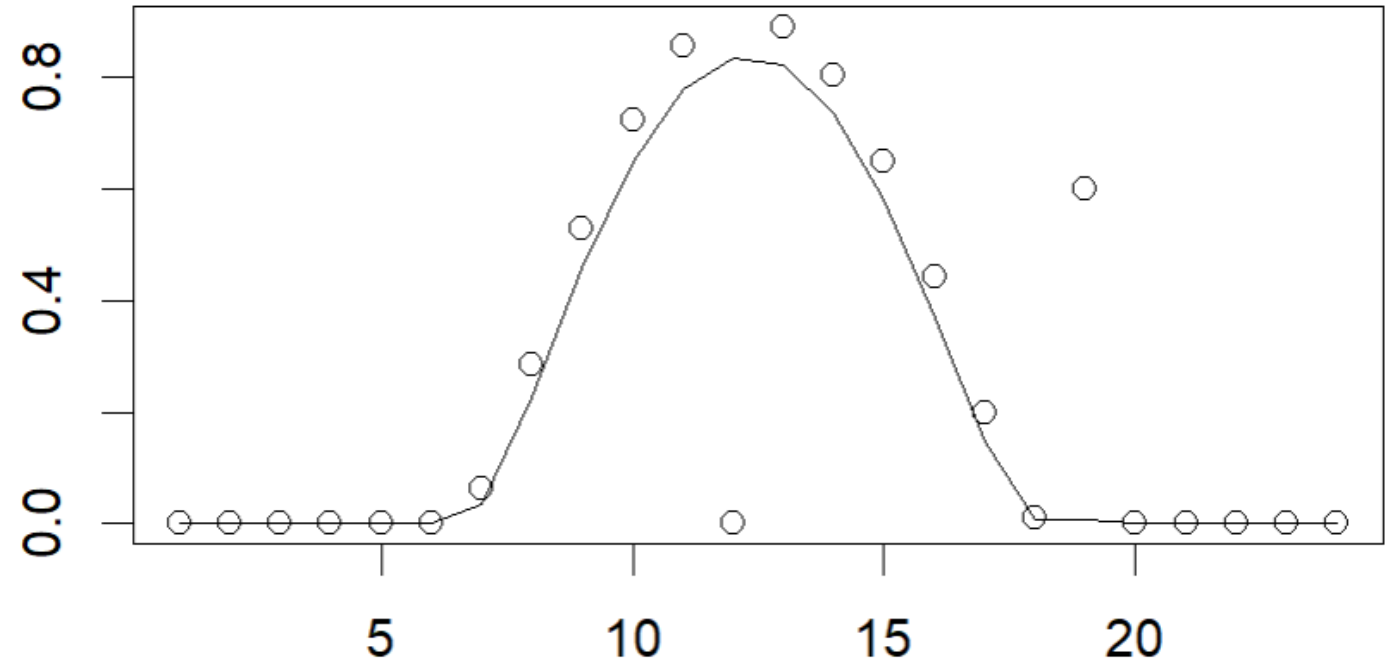
```
i=40  
aux=perfis[indice[i],]  
aux[19]=aux[19]+0.6  
aux[12]=0
```

## # AVALIA A ENTRADA RUIDOSA

```
saida=predict(aeNN,newdata=t(aux))  
maximo=max(c(aux,as.numeric(saida$prediction)))
```

## # GRÁFICO DA ENTRADA RUIDOSA E DA SAÍDA GERADA PELO AUTOENCODER

```
plot(aux,ylim=c(0,maximo)) # entrada ruidosa  
lines(as.numeric(saida$prediction)) # resultado do autoencoder
```





# Autoencoder com o pacote ANN2 do R

## # GERA UMA ENTRADA RUIDOSA

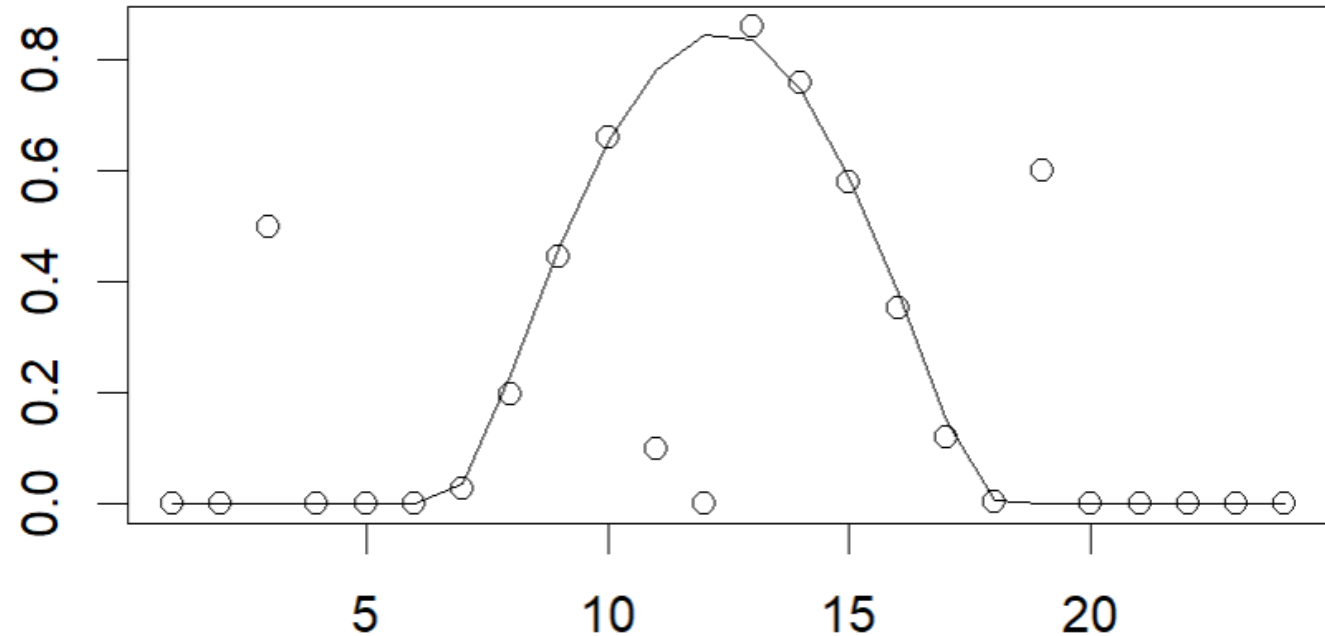
```
i=50  
aux=perfis[indice[i],]  
aux[3]=0.5  
aux[19]=aux[19]+0.6  
aux[11]=0.1  
aux[12]=0
```

## # AVALIA A ENTRADA RUIDOSA

```
saida=predict(aeNN,newdata=t(aux))  
maximo=max(c(aux,as.numeric(saida$prediction)))
```

## # GRÁFICO DA ENTRADA RUIDOSA E DA SAÍDA GERADA PELO AUTOENCODER

```
plot(aux,ylim=c(0,maximo)) # entrada ruidosa  
lines(as.numeric(saida$prediction)) # resultado do autoencoder
```



# Generative Adversarial Network – GAN (2014)



Ian Goodfellow  
1987

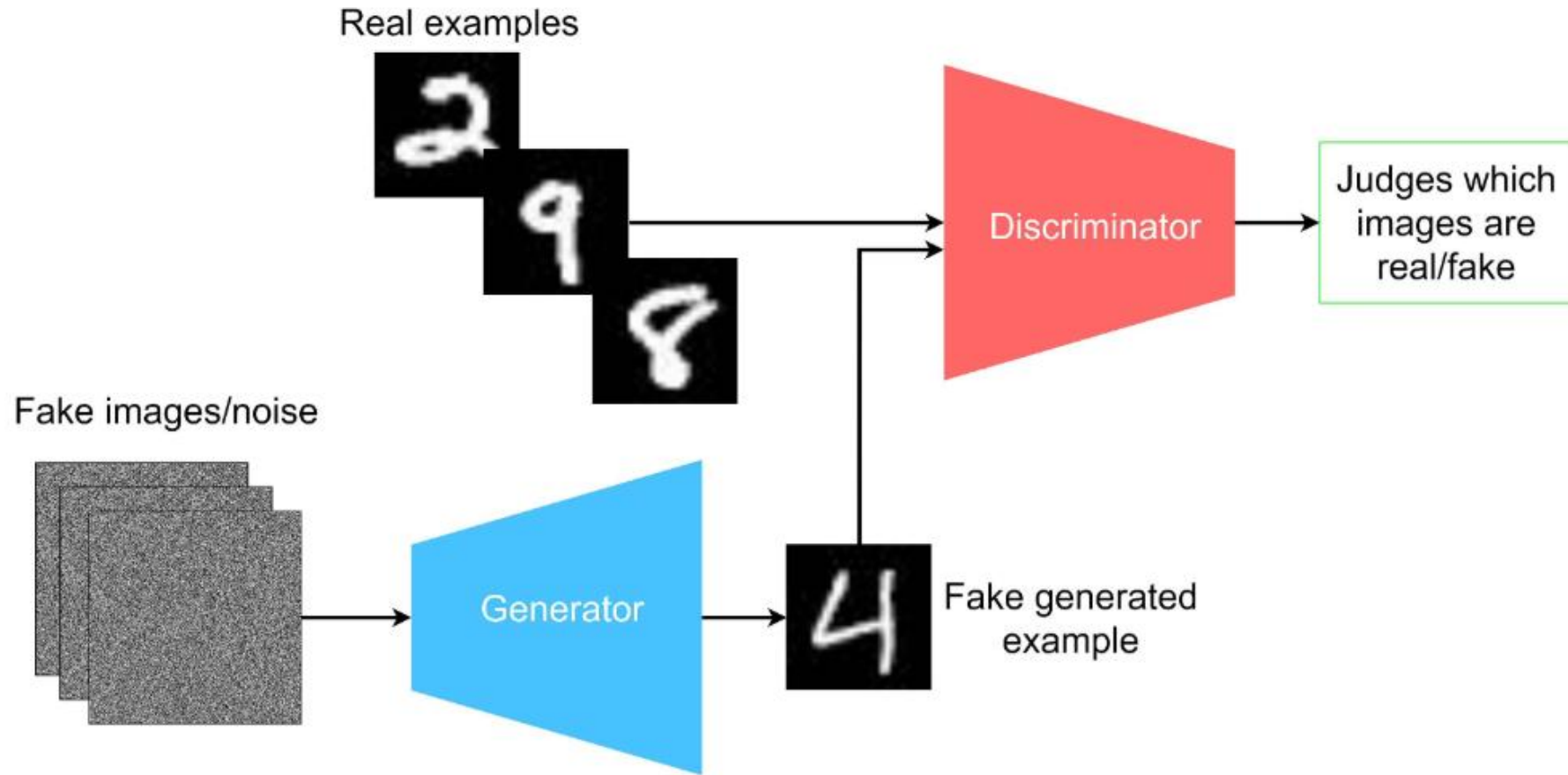
Uma **Generative Adversarial Network (GAN)** é um tipo de rede neural projetada para gerar dados novos que são indistinguíveis de dados reais.

Funciona com base em uma abordagem competitiva entre duas redes neurais: o **gerador (generator)** e o **discriminador (discriminator)**.

O objetivo do treinamento é alcançar um equilíbrio onde:

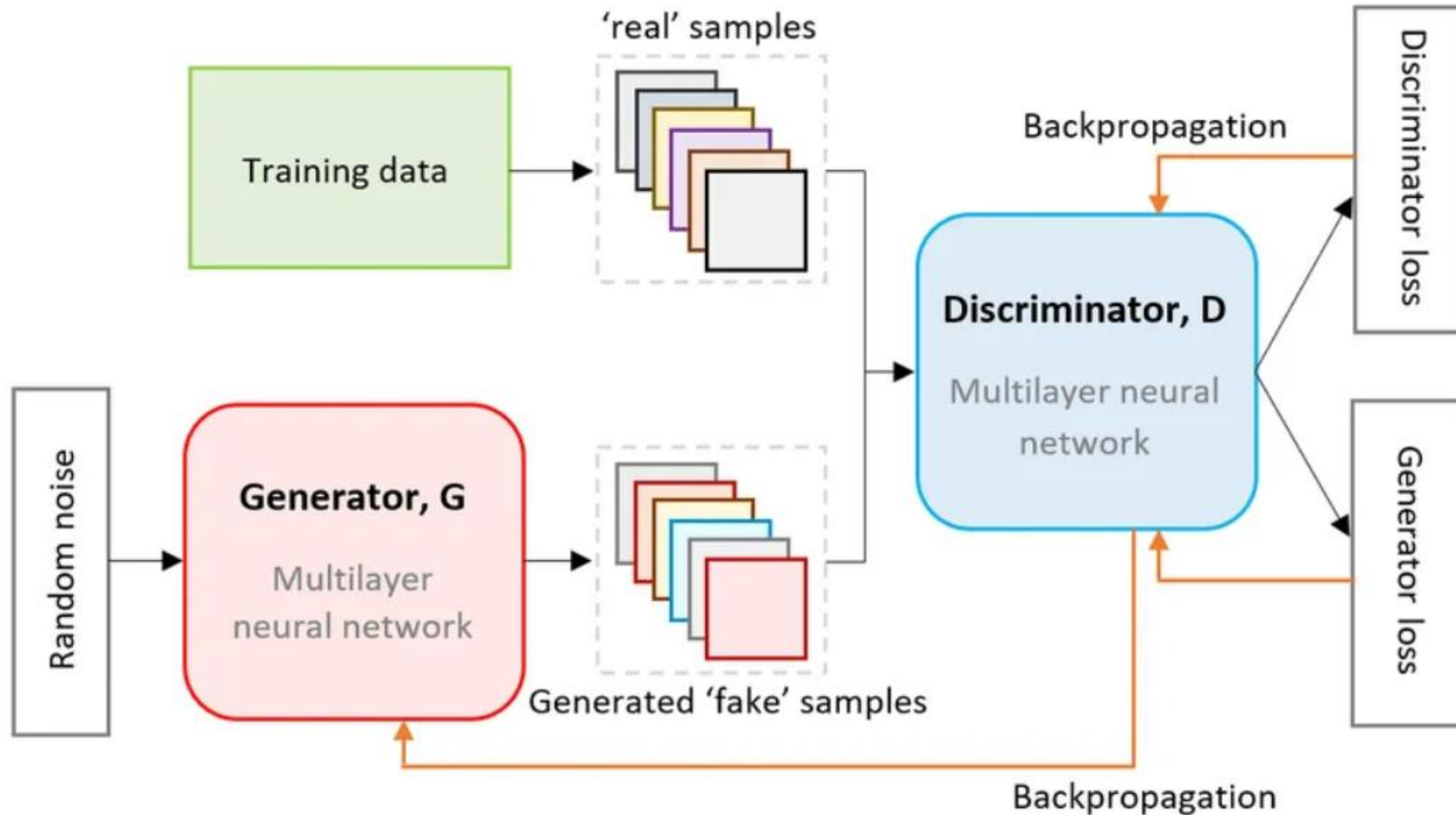
- O gerador aprende a produzir dados realistas.
- O discriminador não consegue distinguir entre os dados reais e os gerados.

# Generative Adversarial Network – GAN (2014)



<https://developer.ibm.com/articles/generative-adversarial-networks-explained/>

# Generative Adversarial Network – GAN (2014)



# Estrutura da GAN

## Estrutura de uma GAN

- **Gerador (Generator):** Tenta criar dados sintéticos que se assemelham aos dados reais.
- Recebe como entrada um vetor de ruído  $z$  (geralmente amostras de uma distribuição aleatória, como uma gaussiana) e o transforma em uma amostra  $G(z)$  que simula os dados reais.
- Seu objetivo é "enganar" o discriminador.

## Discriminador (Discriminator)

- Avalia os dados, classificando-os como reais (provenientes do conjunto de dados original) ou falsos (produzidos pelo gerador).
- Funciona como um classificador binário que tenta distinguir entre  $x$  (dato real) e  $G(z)$  (dato gerado).

## Competição (Adversarial)

- O gerador e o discriminador são treinados simultaneamente em um jogo de soma zero
- O gerador tenta maximizar as chances de o discriminador aceitar suas amostras como reais.
- O discriminador tenta minimizar os erros ao classificar corretamente dados reais e falsos.

# Treinamento da GAN

- O discriminador é treinado para classificar corretamente dados reais e falsos.
- O gerador é treinado para melhorar a qualidade dos dados gerados, "enganando" o discriminador.
- Essas duas redes são atualizadas iterativamente até que o gerador produza dados que o discriminador não consegue distinguir dos reais.

**Matematicamente, o treinamento pode ser formulado como um problema de otimização (jogo minimax):**

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_z} [\log(1 - D(G(z)))]$$

Onde:

$D(x)$ : Probabilidade do discriminador classificar  $x$  como real.

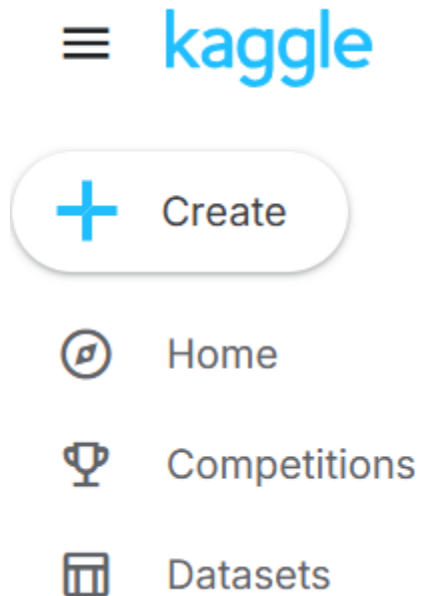
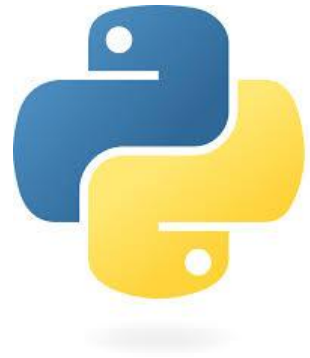
$G(z)$ : Dado gerado pelo gerador a partir do ruído  $z$ .

# Aplicações da GAN

- **Geração de imagens:** Criação de imagens sintéticas realistas (e.g., rostos humanos, paisagens, objetos).
- **Criação de vídeos e animações:** Geração de vídeos realistas ou interpolação de quadros.
- **Dados sintéticos para treinamento:** Gerar dados quando os reais são escassos ou inacessíveis.



# Implementação da rede GAN

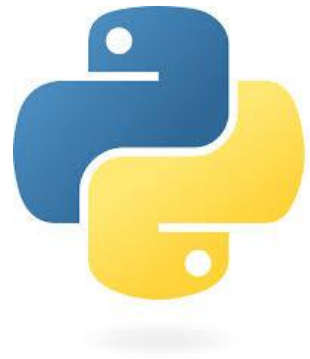


ROHIT@4567 · 10MO AGO · 557 VIEWS

## Time Series GAN with Pytorch

<https://www.kaggle.com/code/rohit4567/time-series-gan-with-pytorch>

# Implementação da rede GAN



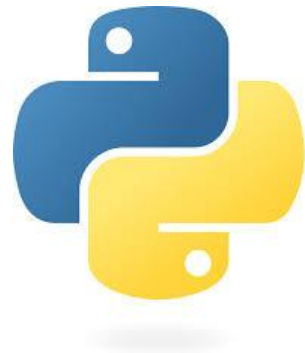
In [2]:

```
import torch
import torch.optim as optim
import torch.nn as nn
from torch.autograd.variable import Variable
import numpy as np
import matplotlib.pyplot as plt
```

In [3]:

```
# Generator some real time-series data
def generate_real_samples(n):
    data = np.random.randn(n)
    return data
```

# Implementação da rede GAN



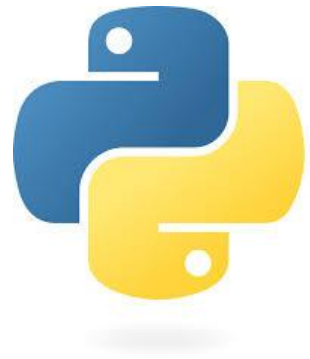
In [4]:

```
# Generator network
class Generator(nn.Module):
    def __init__(self, latent_dim=128):
        super(Generator, self).__init__()
        self.latent_dim = latent_dim
        self.model = nn.Sequential(
            nn.Linear(self.latent_dim, 64), # input dim = latent_dim = (128, 64)
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

    def forward(self, x):
        return self.model(x)
```

## GERADOR

# Implementação da rede GAN



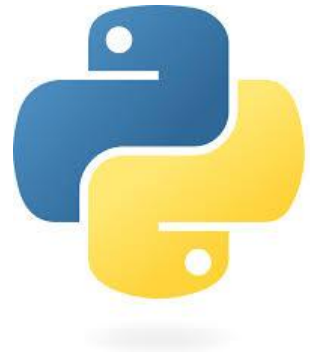
In [5]:

```
# Discriminator Network
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(1, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
            nn.Dropout(p=0.2),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)
```

## DISCRIMINADOR

# Implementação da rede GAN



In [6]:

```
# Function to train the discriminator
def train_discriminator(discriminator, optimizer, real_data, fake_data):
    optimizer_D.zero_grad()

    # Train on real data
    prediction_real = discriminator(real_data)
    error_real = loss(prediction_real, torch.ones_like(prediction_real))
    error_real.backward()

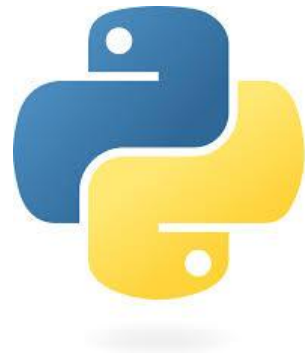
    # Train on fake data
    prediction_fake=discriminator(fake_data.detach())
    error_fake=loss(prediction_fake,torch.zeros_like(prediction_fake))
    error_fake.backward()

    optimizer_D.step()

    return error_real+error_fake
```

**TREINA O DISCRIMINADOR**

# Implementação da rede GAN



In [7]:

```
# Function to train the generator  
def train_generator(generator, optimizer, fake_data):  
    optimizer_G.zero_grad()  
  
    prediction = discriminator(fake_data)  
    error = loss(prediction, torch.ones_like(prediction))  
    error.backward()  
  
    optimizer_G.step()  
  
    return error
```

**TREINA O GERADOR**

# Implementação da rede GAN

*# Training loop*

**for** epoch **in** range(1, epochs+1):

*# Generate real and fake data*

real\_data = torch.Tensor(generate\_real\_samples(batch\_size)).view(-1, 1)

fake\_data = generator(Variable(torch.randn(batch\_size, latent\_dim)))

*# Train discriminator*

d\_loss = train\_discriminator(discriminator, optimizer\_D, real\_data, fake\_data)

*# Train generator*

g\_loss = train\_generator(generator, optimizer\_G, fake\_data)

**if** epoch % 100 == 0:

**print**(f"Epoch: {epoch}, D Loss: {d\_loss.item()}, G Loss: {g\_loss.item()}")

## LOOP DE TREINAMENTO

*# Hyperparameters*

batch\_size = 128

lr = 3e-4

epochs = 5000

*# Models and optimizer*

generator = Generator()

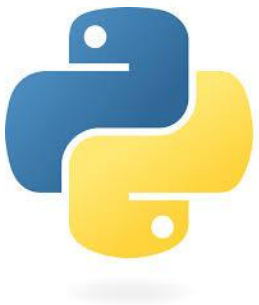
discriminator = Discriminator()

optimizer\_G = optim.Adam(generator.parameters(), lr=lr)

optimizer\_D = optim.Adam(discriminator.parameters(), lr=lr)

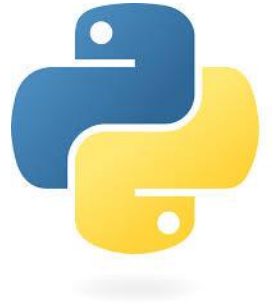
loss = nn.BCELoss()

latent\_dim = 128





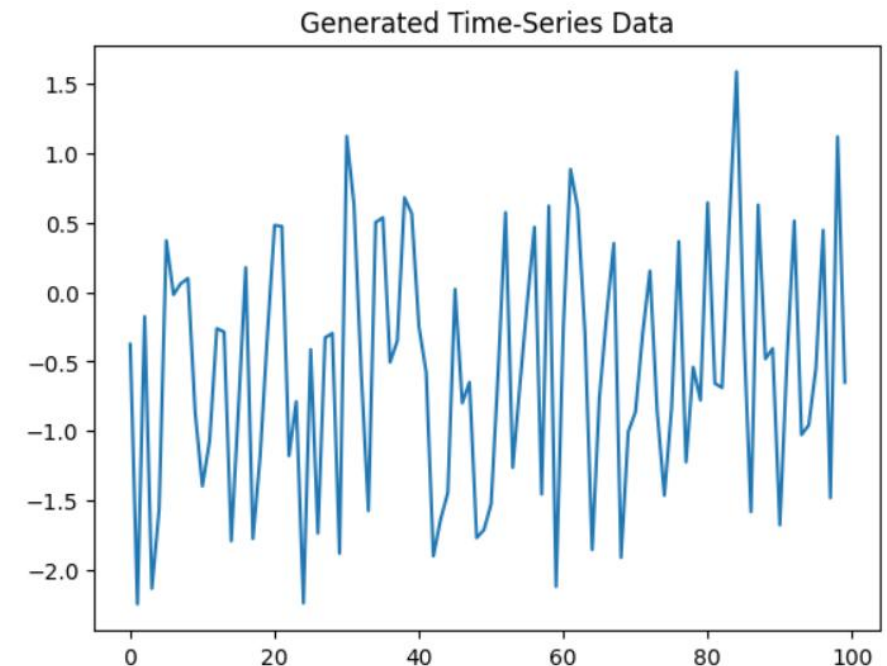
# Implementação da rede GAN



In [9]:

```
# Generate synthetic time-series data using the trained generator  
generated_data = generator(Variable(torch.randn(100, latent_dim))).detach().numpy()  
  
# Plot the generator  
plt.plot(generated_data)  
plt.title("Generated Time-Series Data")  
plt.show()
```

**GERA SÉRIE SINTÉTICA**



# Considerações finais

**Foram apresentadas algumas arquiteturas de redes neurais artificiais para treinamento não supervisionado: SOM, Autoencoder e GAN**

**Aplicações apresentadas: análise de agrupamentos, detecção de anomalias e geração de dados sintéticos.**

**Os recursos disponibilizados pelo R e Python facilitam a implementação das redes neurais artificiais**

**Os códigos disponibilizados na apresentação podem ser facilmente adaptados para outras aplicações.**

# Referências bibliográficas

**KELLEHER, J.D. Deep Learning, MIT Press, 2019.**