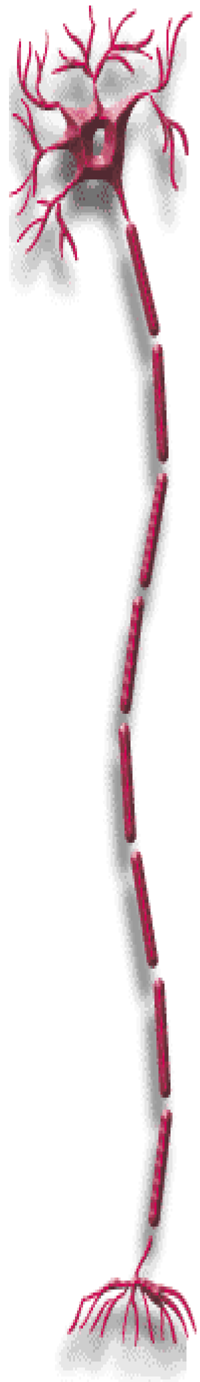
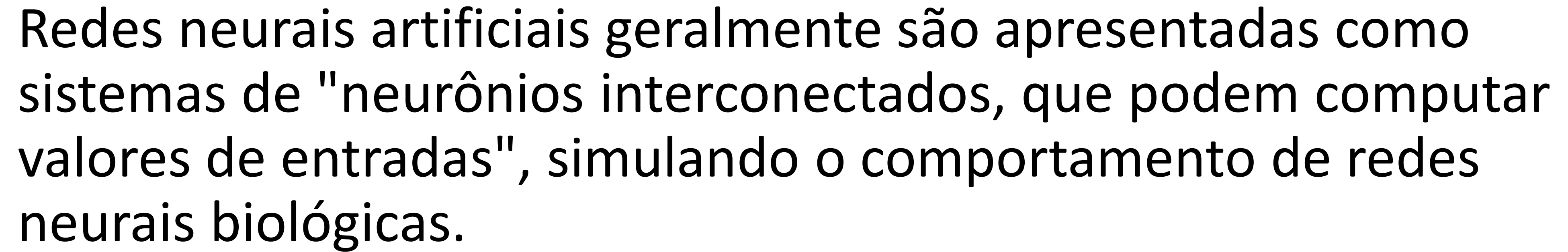


Programa de Pós-Graduação em Engenharia de Recursos Hídricos e Ambiental

PPGERHA

Machine Learning
Deep Learning
Neural nets

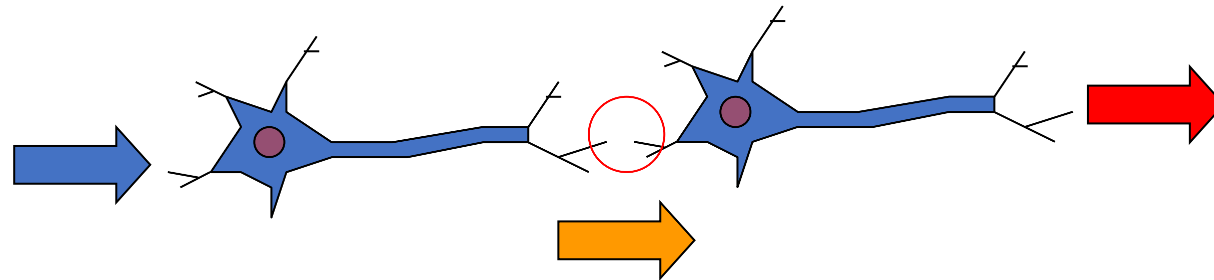
Jorge Centeno



Redes neurais artificiais geralmente são apresentadas como sistemas de "neurônios interconectados, que podem computar valores de entradas", simulando o comportamento de redes neurais biológicas.

sinapse

O sinal nervoso (impulso), que vem através do axônio da célula pré-sináptica chega em sua extremidade e provoca a liberação de ***neurotransmissores***. Este elemento químico se liga quimicamente a ***receptores*** específicos no neurônio pós-sináptico, dando continuidade à propagação do sinal.

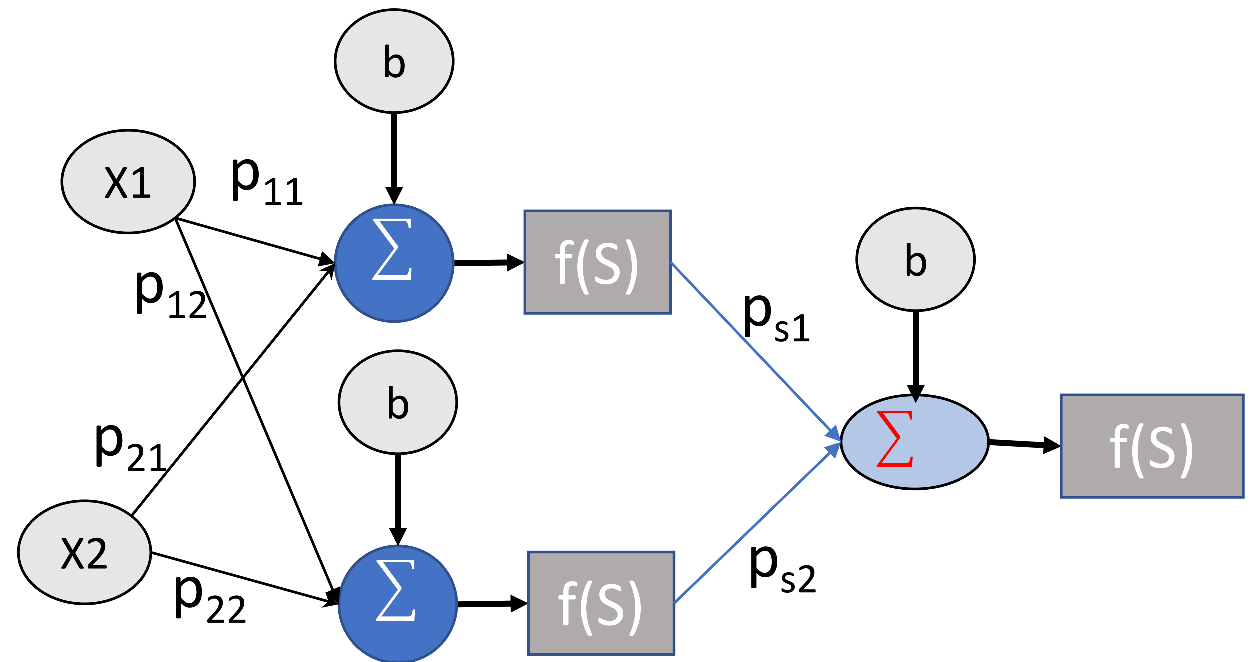


Modelo matemático

O mesmo ocorre em uma rede neural artificial como o multilayer perceptron.

Neste caso, o sinal de entrada pode (ou não) gerar sinais de saída que são (ou não) repassados para um elemento seguinte.

Aqui, os pesos e a função de transferência jogam um papel determinante.

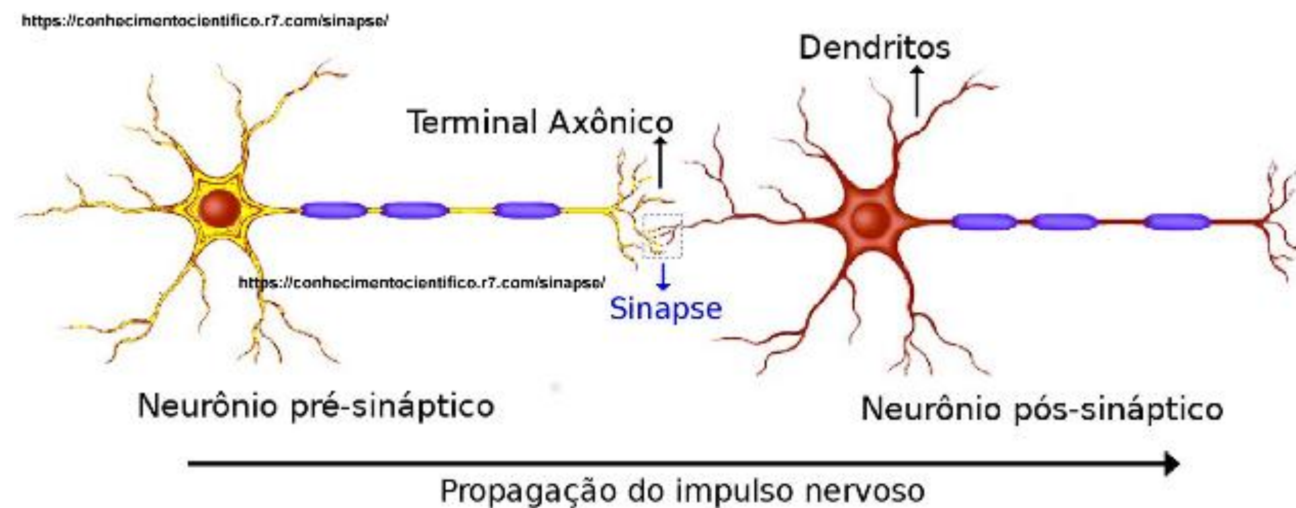


Sinapse

No sistema nervoso, os impulsos devem passar de uma célula à outra para que ocorra uma resposta a um determinado sinal.

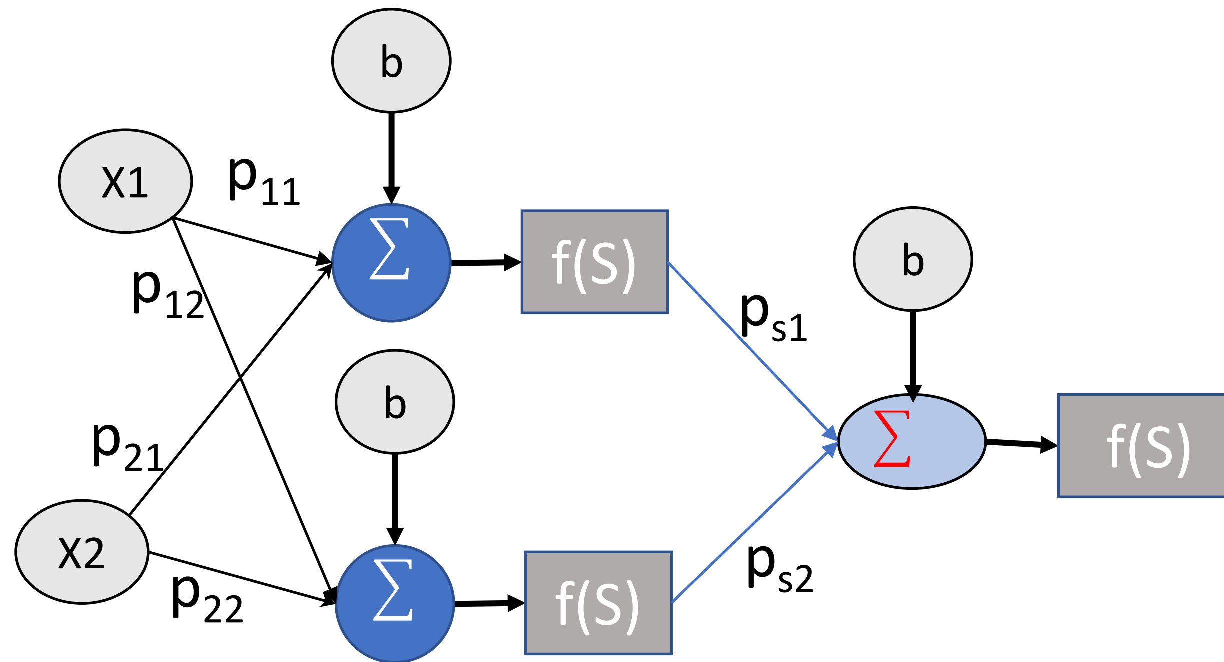
Isto ocorre na sinapse suma região de proximidade entre a extremidade de um neurônio e uma célula vizinha, onde os impulsos nervosos são transformados em impulsos químicos em decorrência da presença de mediadores químicos.

As sinapses podem ativar o outro neurônio ou passar um sinal de inibição



Sinapse

Em nosso caso, os pesos Ps1 e Ps2 fazem o papel da sinapse, pois repassam ou não a saída dos neurônios para o elemento de saída.



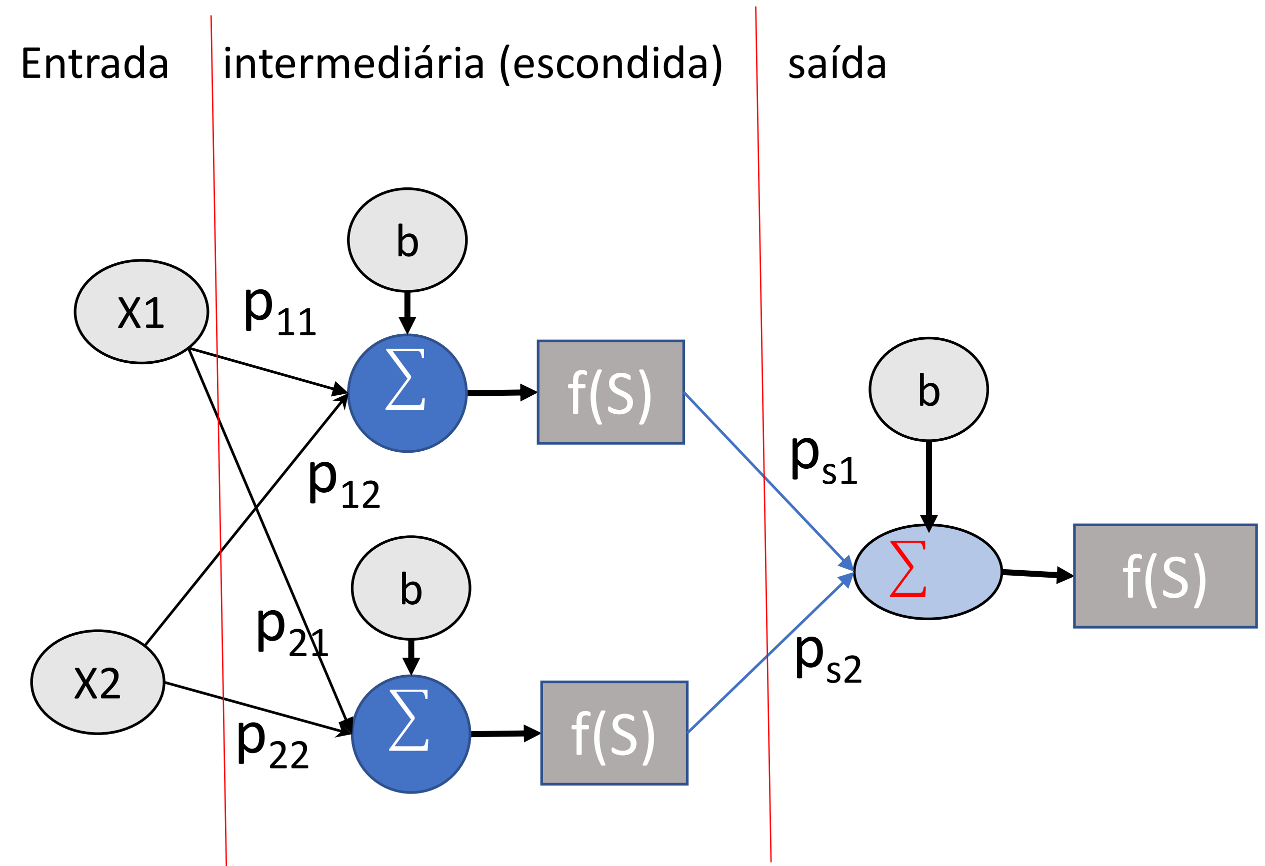
Camadas

Então, nossa pequena rede pode ser dividida em três camadas:

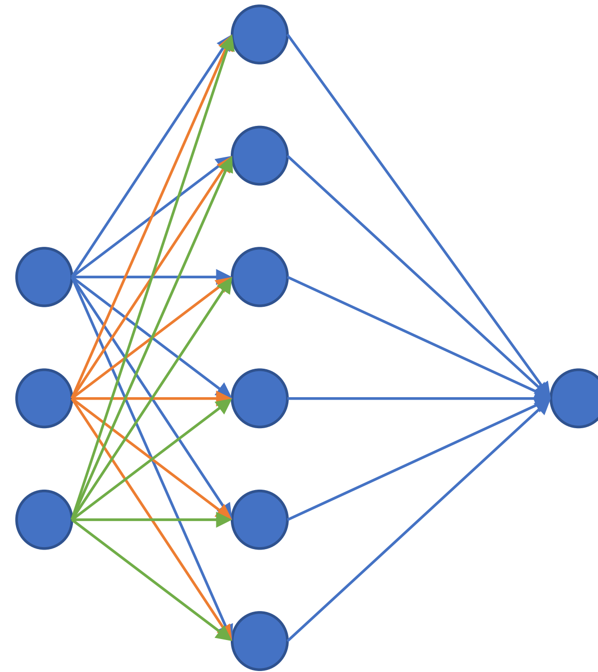
a) Camada de entrada (não tem pesos, apenas recebe as entradas)

b) Camada intermediária (composta pelos neurônios em paralelo)

c) Camada de saída



Podemos aumentar o número de neurônios na camada escondida...

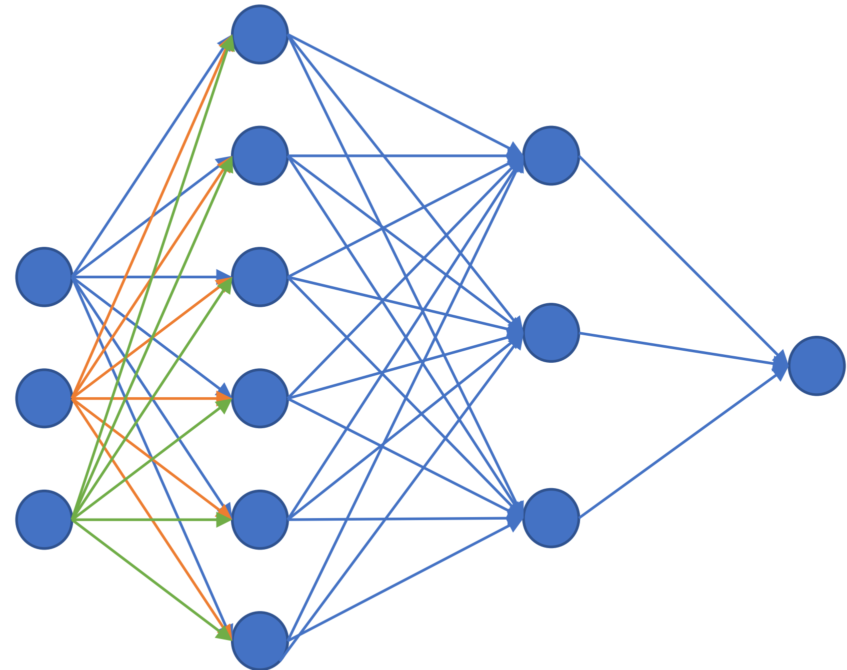


Ou usar mais de uma camada escondida...

Com número de elementos diferentes.

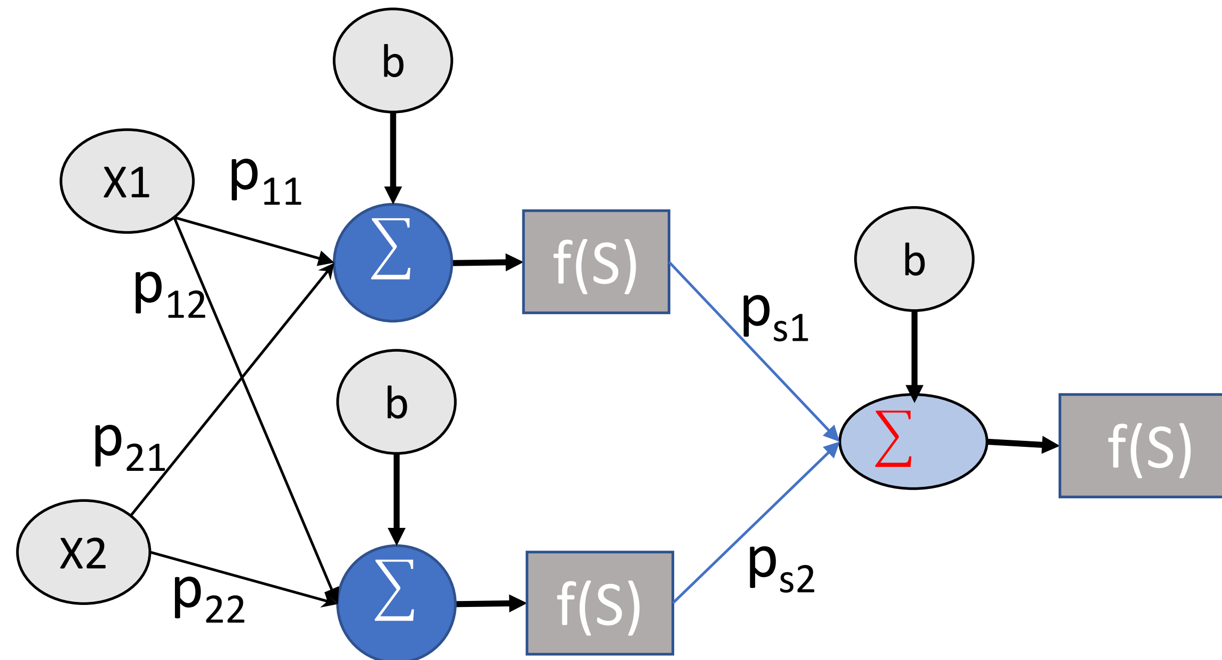
As possibilidades são imensas

A quantidade de pesos também ...



Função de ativação

Até o momento usamos uma função degrau para calcular a saída (zero ou um) de um perceptron. Porém, a saída também poderia ser um outro valor, um caso mais geral, se outra função de ativação for usada.



Regra de Hebb (1949)

Usada como base para introduzir o aprendizado no perceptron e em redes neurais.

Regra de Hebb

SE

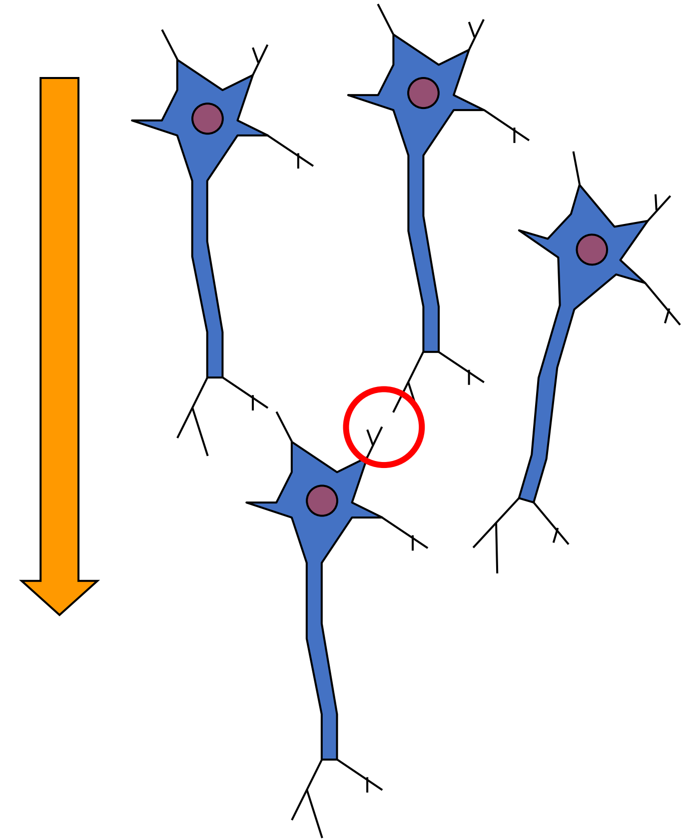
O axonio de uma célula “A” se encontrar suficientemente próximo de outra célula “B” como para exitá-la

E

Existe ativação frequente (repetitiva) e persistente entre estas células

Então:

Ocorre algum tipo de processo de crescimento ou alteração metabólica em uma ou ambas células, de modo que aumenta a eficiência da conexão entre A e B.



Regra de Hebb

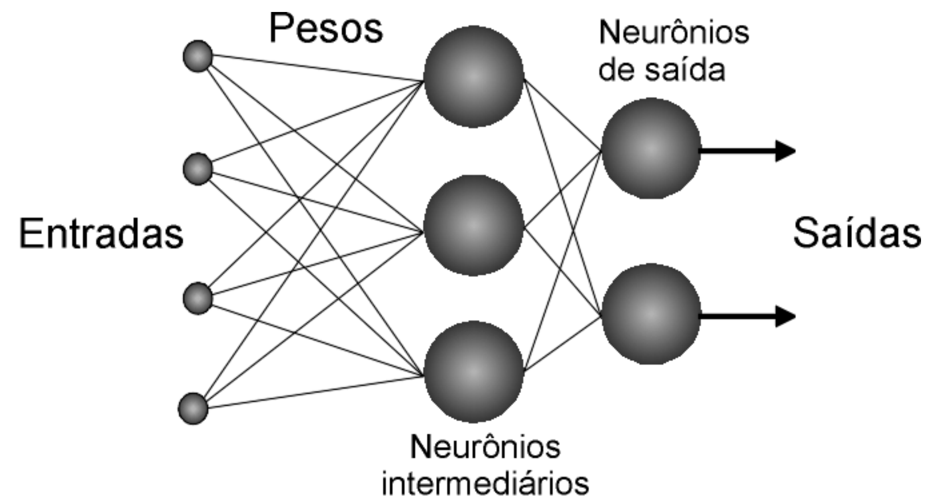
Se um caminho é possível e muito utilizado, então o peso desse caminho é reforçado.

Pelo contrario, se uma determinada conexão não é ativada frequentemente, seu peso tende a diminuir.

Aprendizado

Todo ser vivo dotado de um sistema nervoso é capaz de modificar o seu comportamento em função de experiências passadas. Essa modificação comportamental é chamada de aprendizado.

Em uma rede neural artificial, o aprendizado consiste em adaptar os valores dos pesos, que controlam a intensidade do sinal de entrada a cada neurônio. Ajustando os pesos, a rede é treinada para responder seguindo um padrão.



ADALINE

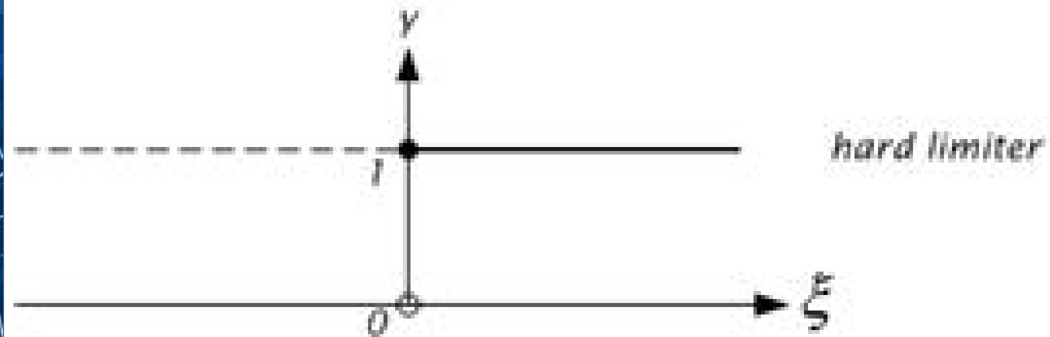
ADALINE (Adaptative Linear Neuron) Widrow e Hoff, 1959.

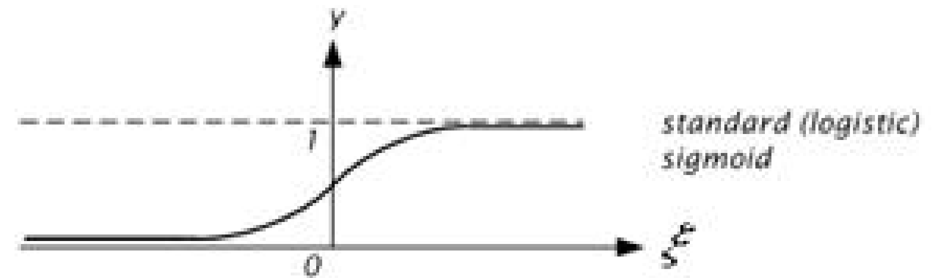
É uma evolução do modelo de neurônio matemático que permite saídas tanto discretas (zero, um) quanto contínuas e pode ser usado para tarefas de classificação e regressão.

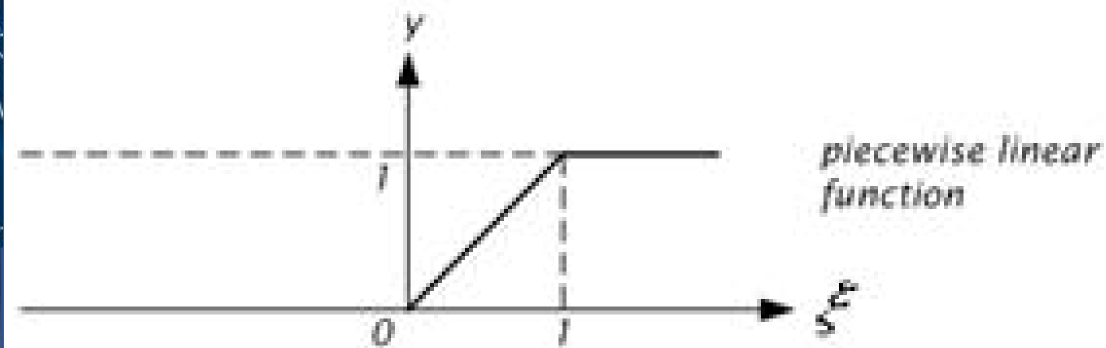
Isto é possível usando funções de transferência contínuas, que produzem saídas contínuas lineares ou não.

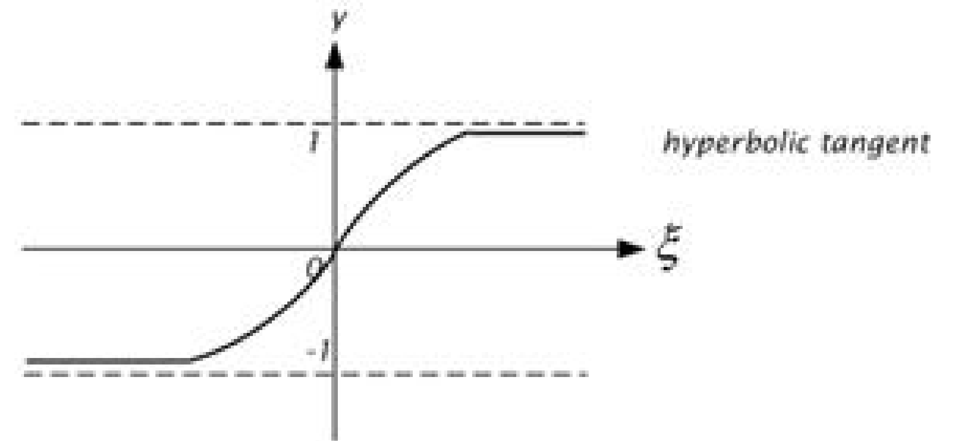
Opções

Outras opções podem ser aplicadas para a modulação as saída









ADALINE: Treinamento

Perceptron ajusta os pesos somente quando um padrão é classificado incorretamente (erro)

O Adaline é similar ao Perceptron, mas usa um algoritmo de treinamento mais refinado, baseado no método dos mínimos quadrados que busca minimizar o erro global das estimativas

Adaline utiliza a **regra Delta** para minimizar o erro médio (MSE) após cada padrão ser apresentado, ajustando os pesos proporcionalmente ao erro.

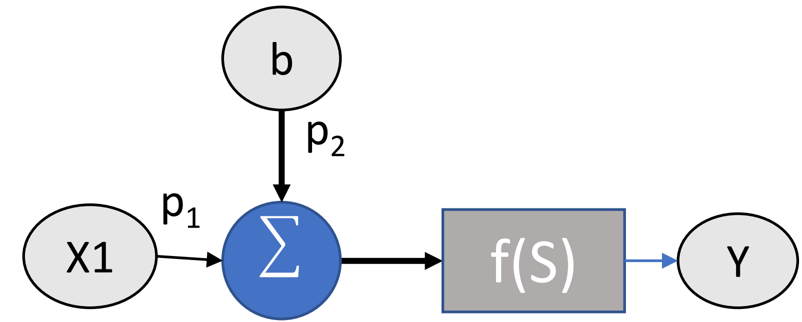
regra Delta

A regra Delta é eficiente quando são apresentados dados não linearmente separáveis.

Ela converge até um valor desejado aproximado, onde a função tem taxa de variação máxima, analisando a variação local do gradiente da função de erro global.

Este algoritmo pode ser visto como uma "caminhada" no domínio da função do erro , em que casa passo é feito no sentido oposto ao gradiente da função no ponto atual.

Adaline: ilustrativo



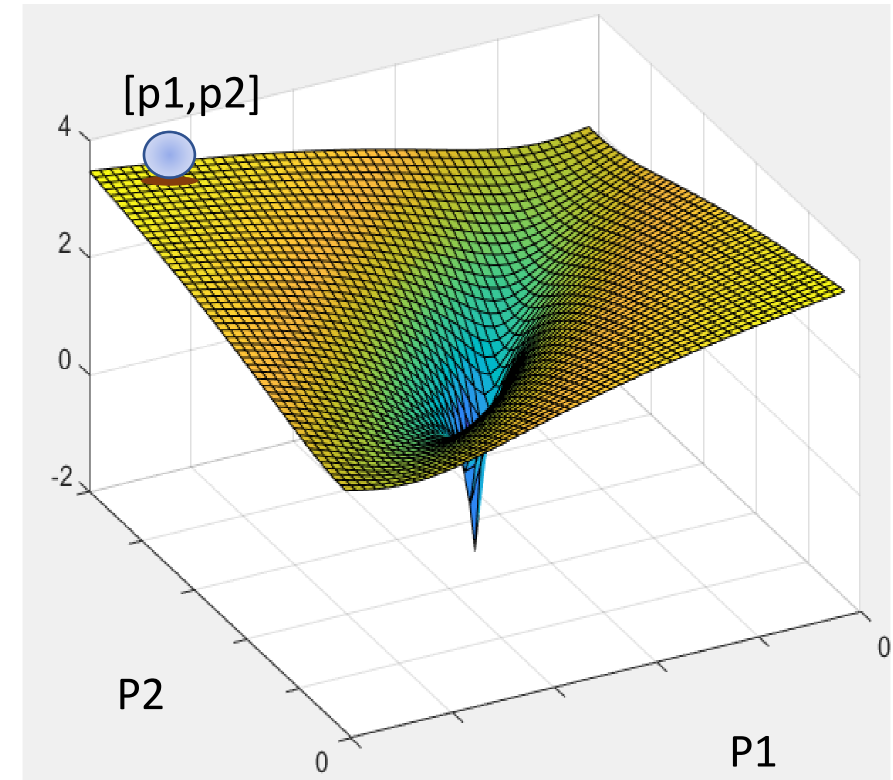
Considere um neurônio com apenas uma entrada (X) e uma saída (Y):

O problema aqui se resume a ajustar dois pesos (p1 para X e p2 para b)

Se conseguirmos mapear todas as possíveis combinações de p1 e p2 poderíamos mapear a superfície de erro.

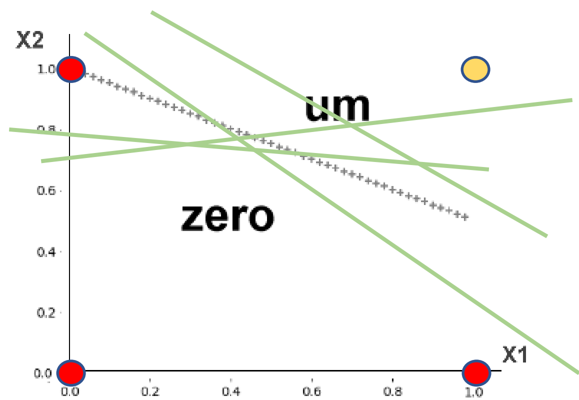
O ideal seria deslocar a solução para o mínimo, mesmo que outras regiões próximas já permitam obter uma solução viável.

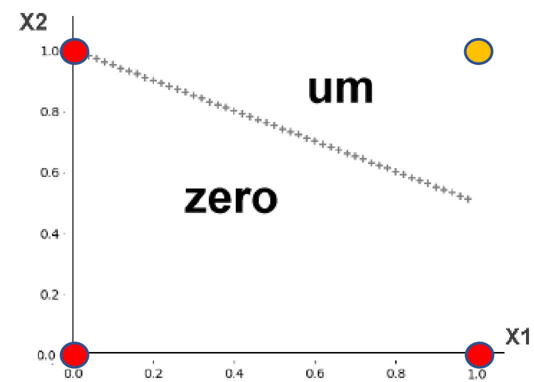
A função de custo a ser minimizada é a soma dos erros ao quadrado:



$$E = \sum_i (Y_i - Ys_i)^2$$

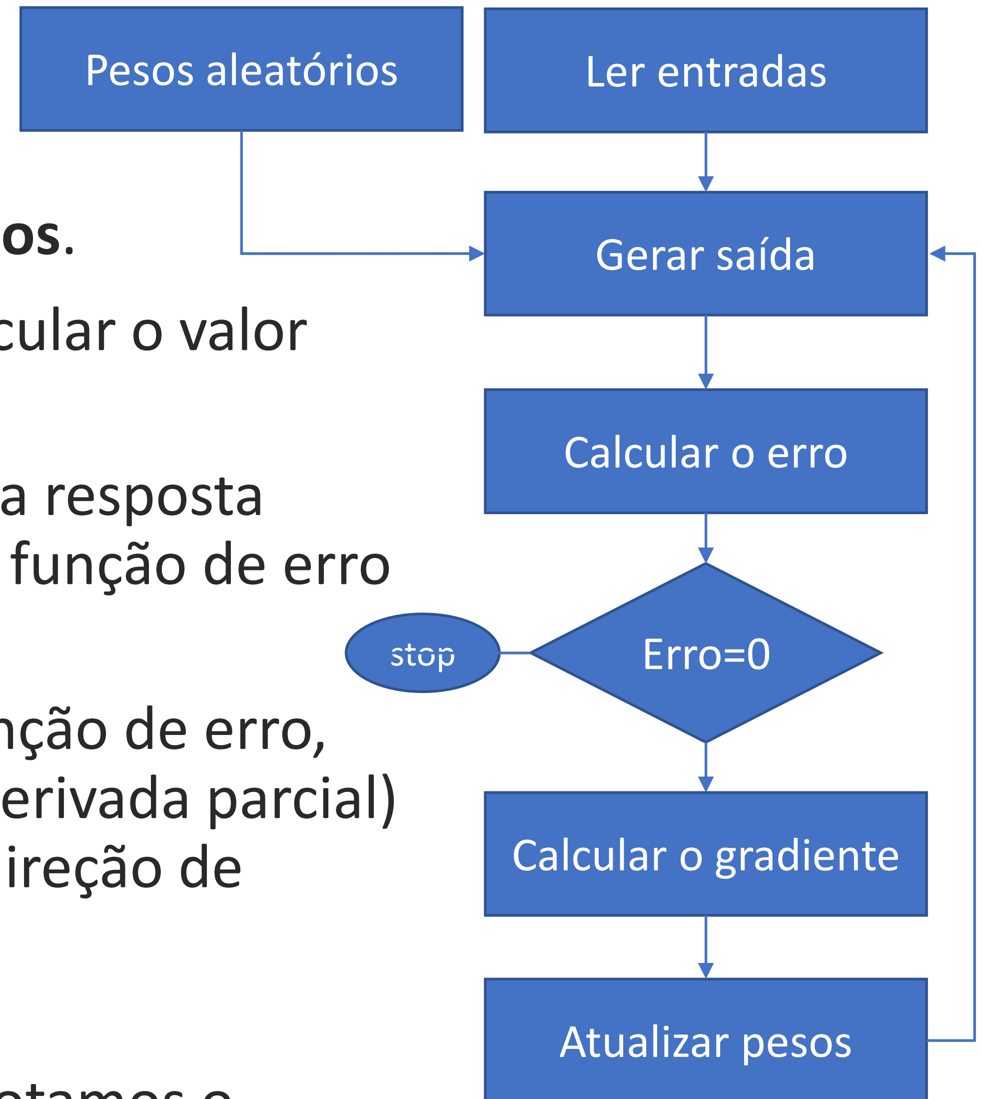
Outras soluções viáveis para nosso velho problema





Para isto...

- Inicializar os pesos com valores **aleatórios**.
- Fornecer dados de entrada à rede e calcular o valor da **função** de erro.
- Como o aprendizado é supervisionado, a resposta correta é conhecida. É importante que a função de erro seja **diferençável**.
- Na tentativa de minimizar o valor da função de erro, calculam-se os valores dos **gradientes** (derivada parcial) para cada peso da rede. Para estimar a direção de maior **crescimento** da função do erro.
- Como se deseja andar na direção de maior **decréscimo** da função de erro, adotamos o sentido contrário ao do gradiente.



Aprendizado: correção dos pesos

O valor do peso atualizado é o valor do peso na iteração anterior, corrigido por um valor proporcional ao gradiente, na direção contrária à do gradiente;

O parâmetro η representa a taxa de aprendizado da, controlando a tamanho do passo que tomamos na correção do peso.

$$p_i(t+1) = p_i\,(t) - \eta\,\frac{\delta E}{\delta p_i}$$

Obs: gradiente positivo: erro aumenta!

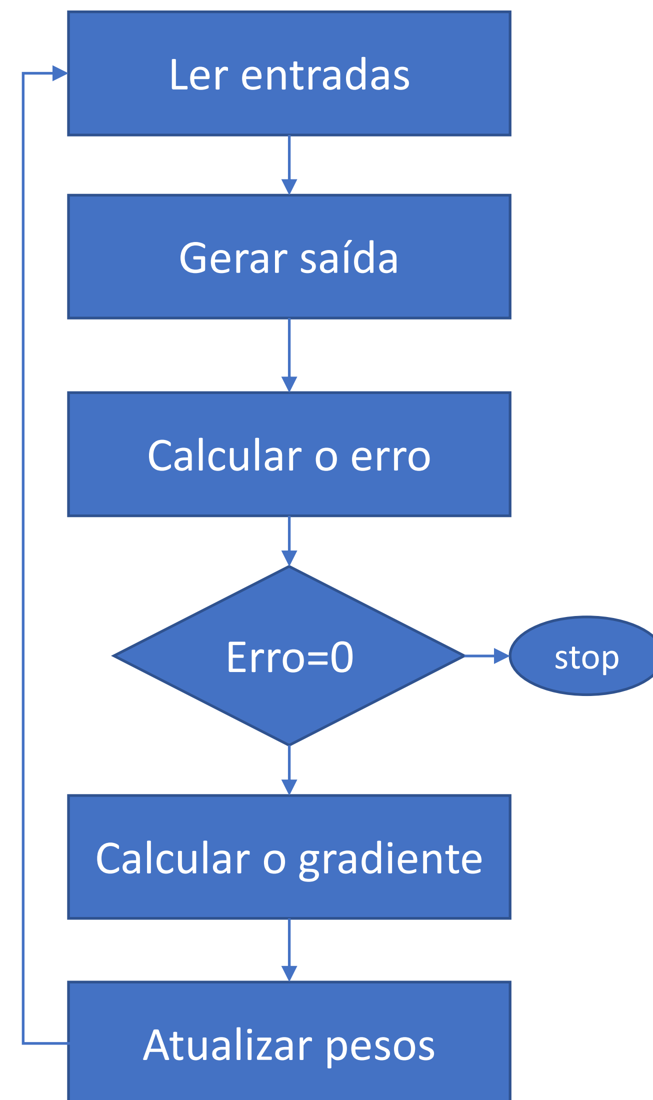
Processo iterativo

Uma vez calculado o vetor gradiente, cada peso é atualizado modo **iterativo**, recalculando o gradiente em cada passo de iteração, até o erro diminuir e chegar a zero ou um limiar preestabelecido, ou o número de iterações atingir um valor máximo.

Mas como calcular o gradiente em relação a cada peso e o bias?

$$\frac{\delta E}{\delta p} = ?$$

$$\frac{\delta E}{\delta b} = ?$$



https://medium.com/ensina-ai/redes-neurais-perceptron-multicamadas-e-o-algoritmo-backpropagation-eaf89778f5b8

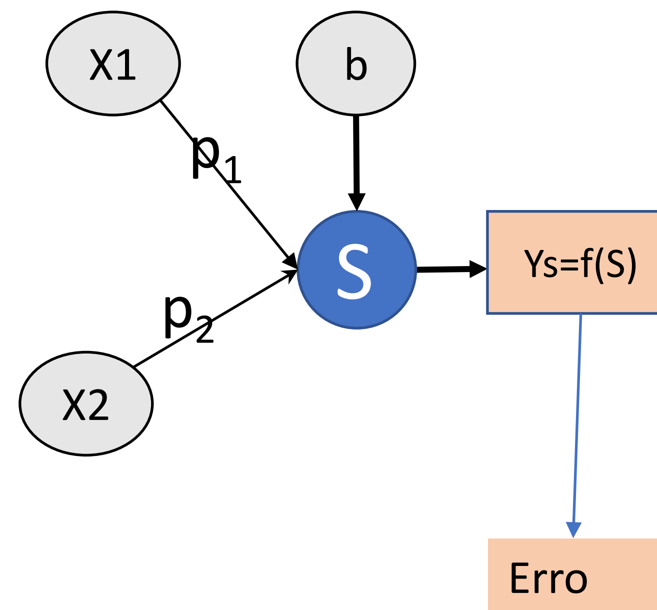
Queremos d(erro)/d(peso)

Mas conhecemos que

$$E = \sum_i (Y_i - Ys_i)^2$$

$$Ys = \sigma(S) = \frac{1}{1 + e^S}$$

S = b + p1*x1 + p2*X2 + … + pn*Xn



Pela regra da cadeia:

$$\frac{\delta E}{\delta p_i} = \frac{\delta E}{\delta Ys} * \frac{\delta\, Ys}{\delta S} * \frac{\delta\, S}{\delta p_i}$$

Começando pela derivada de E em relação a Ys

$$\frac{\delta E}{\delta Ys} = -2(Y_i - Ys_i)$$

A saída é função da soma ponderada das entradas Ys= f(S)

Ex: função sigmoide

$$Ys = \sigma(S) = \frac{1}{1 + e^{S}}$$

Cuja derivada em relação a S é

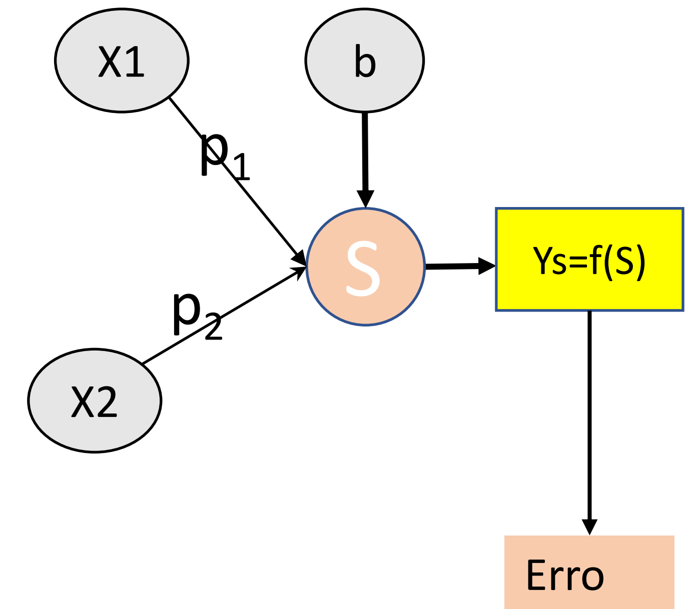
$$\frac{\delta\, Ys}{\delta x} = \frac{\delta\, \sigma(x)}{\delta x} = \sigma(S)(1-\sigma(S))$$

Pela regra da cadeia a derivada parcial do erro em relação à saída do neurônio é:

$$\frac{\delta E}{\delta S} = \frac{\delta E}{\delta Ys} * \frac{\delta Ys}{\delta S} = -2(Y_i - Ys_i)\sigma(S)(1-\sigma(S))$$

Chama-se isto de delta Δ

$$\Delta = \frac{\delta E}{\delta Ys} * \frac{\delta Ys}{\delta S} - 2(Y_i - Ys_i)\sigma(S)(1-\sigma(S))$$



Como “S” é a soma das entradas multiplicadas pelos respectivos pesos

S = b + p1*x1 + p2*X2 + ... + pn*Xn

A derivada de “S” em relação ao peso “p” da i-éssima entrada é

$$\frac{\delta S}{\delta p_i} = X_i$$

E do bias:

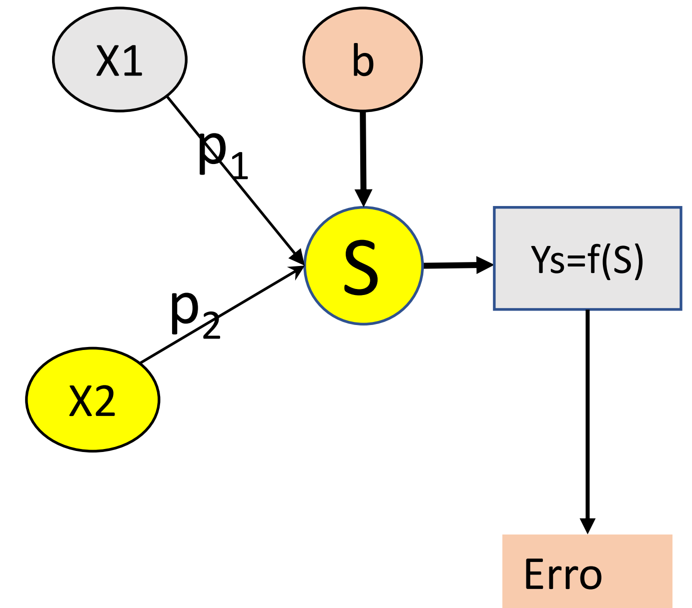
$$\frac{\delta S}{\delta\, b} = 1$$

Então, pela regra da cadeia:

$$\frac{\delta E}{\delta p_i} = \frac{\delta E}{\delta Ys} * \frac{\delta\, Ys}{\delta S} * \frac{\delta\, S}{\delta p_i}$$

ou $$\frac{\delta E}{\delta p_i} = \Delta * \frac{\delta\, S}{\delta p_i}$$

$$\frac{\delta E}{\delta b} = \Delta$$

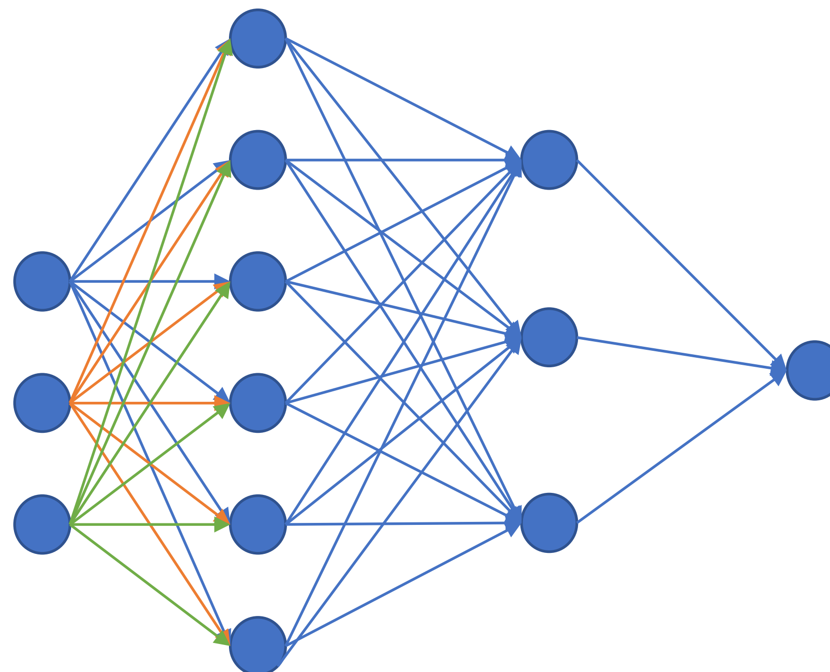


Feedforward/back propagation

Então:

1. Fornecer entradas
2. Propagar resultados para frente
3. Obter saídas
4. Calcular erro
5. Propagar o erro de volta

Até atingir o erro mínimo estabelecido ou o número de iterações máximo

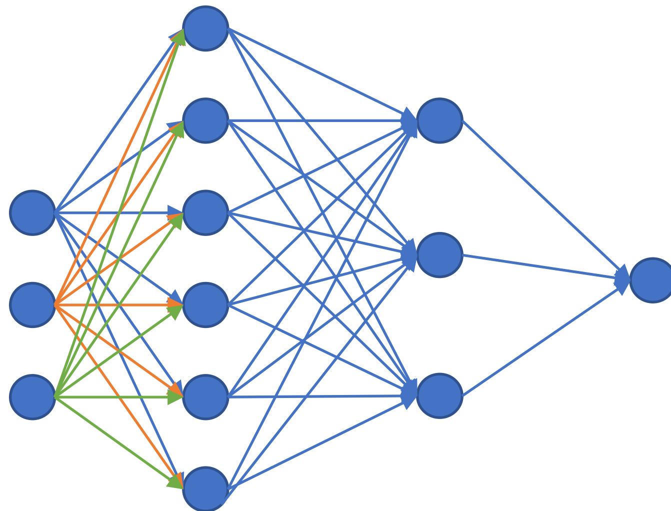


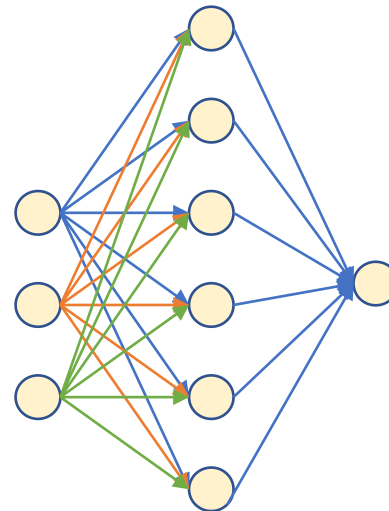
Arquitetura da rede

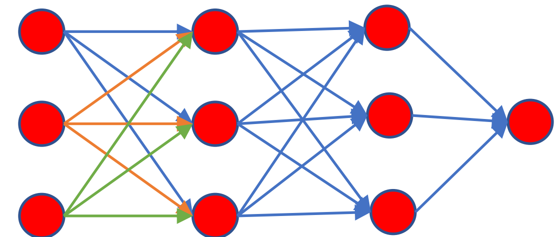
Não existe uma regra geral para determinar o número de camadas ocultas e neurônios;
Isto é determinado, geralmente, por tentativa e erro

O número de neurônios da camada oculta influencia significativamente o desempenho de uma red.

Um número pequeno: a rede pode não atingir o nível desejado de acurácia
Muitos neurónios: O treinamento se torna lento e se corre o risco de overfitting.







Um exemplo

International Journal of Hybrid Information Technology
Vol.9, No.3 (2016), pp. 263-272
http://dx.doi.org/10.14257/ijhit.2016.9.3.24

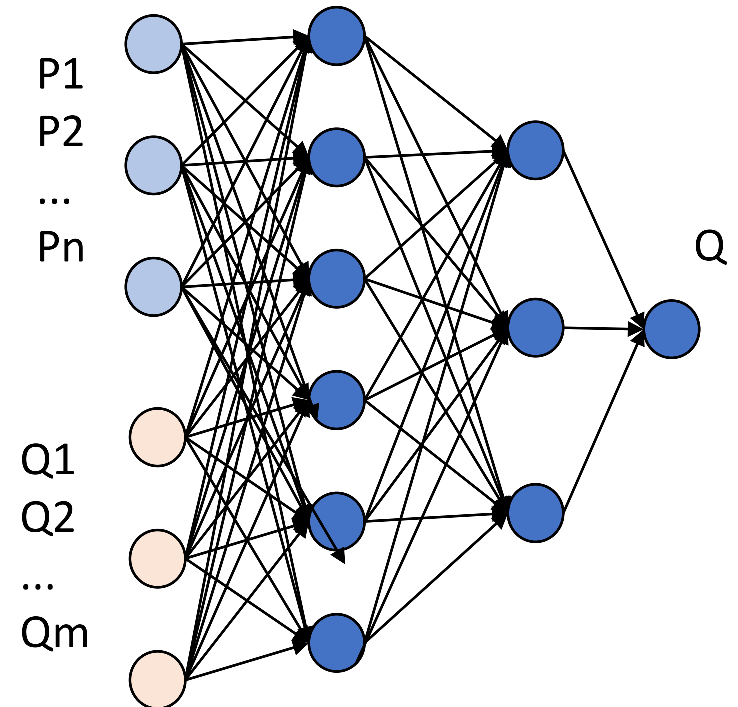
Artificial Neural Network Model for Rainfall-Runoff -A Case Study

P.Sundara Kumar, T.V.Praveen and M. Anjanaya Prasad

Entradas: séries diárias de precipitação e vazão.

Saída: vazão

Como entrada, usa-se uma série de (N) dados de chuva e (M) de vazão.



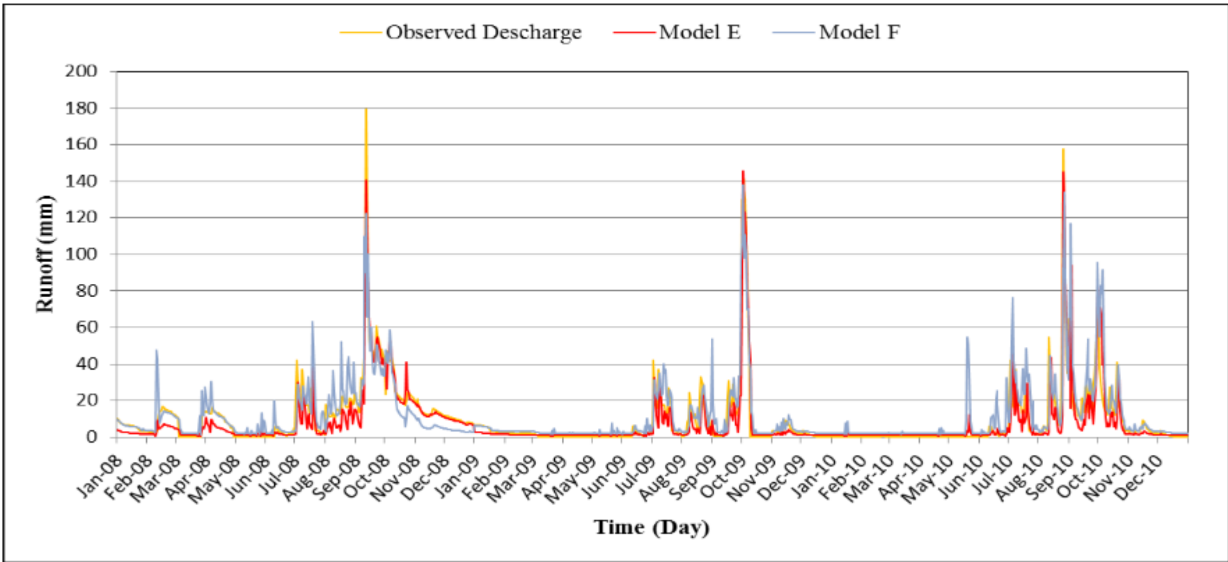
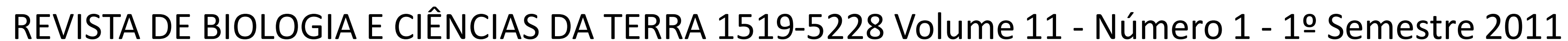


Figure 7. Comparison of Runoff Values between Observed and ANN Model during Validation Period


REVISTA DE BIOLOGIA E CIÊNCIAS DA TERRA 1519-5228 Volume 11 - Número 1 - 1º Semestre 2011

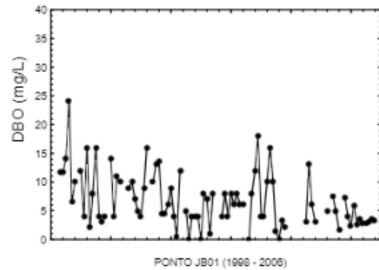
Redes neurais em análises de qualidade de água

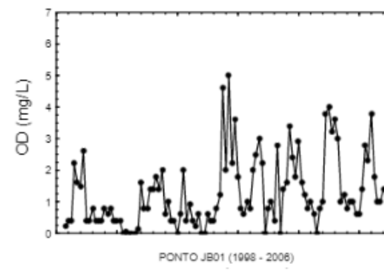
Daniel Christiano; Malva Isabel Medina Hernández ; Pedro L. Christiano

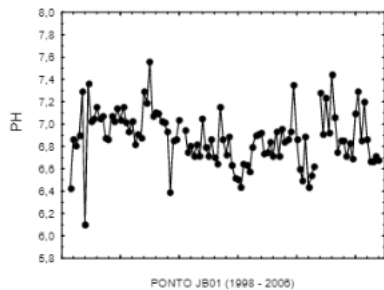
Foram usadas séries temporais de: o nível de coliformes fecais, OD, DBO e pH para predizer estas variáveis no futuro.

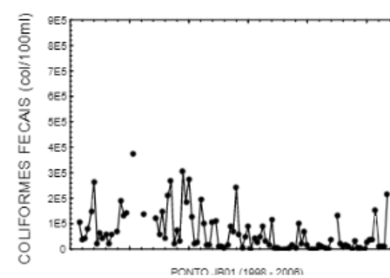
Exemplo

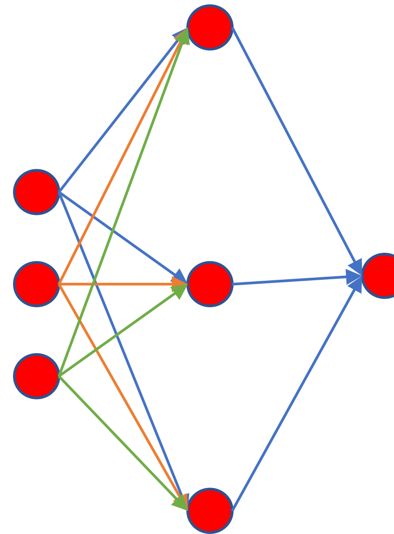
coliformes fecais, OD, DBO e pH

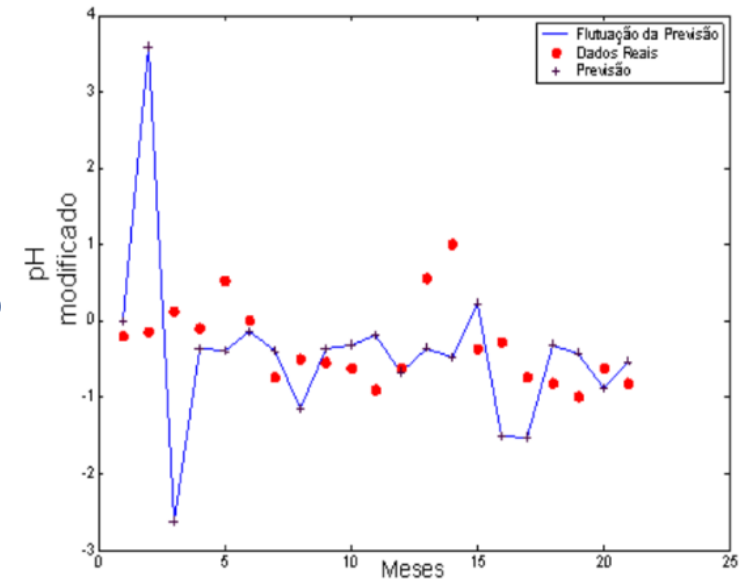












Classificação

Dadas três variáveis de qualidade de água, deseja-se enquadrar os corpos de água em quatro classes possíveis de qualidade:

Entradas: 3

Saída 1, porém...

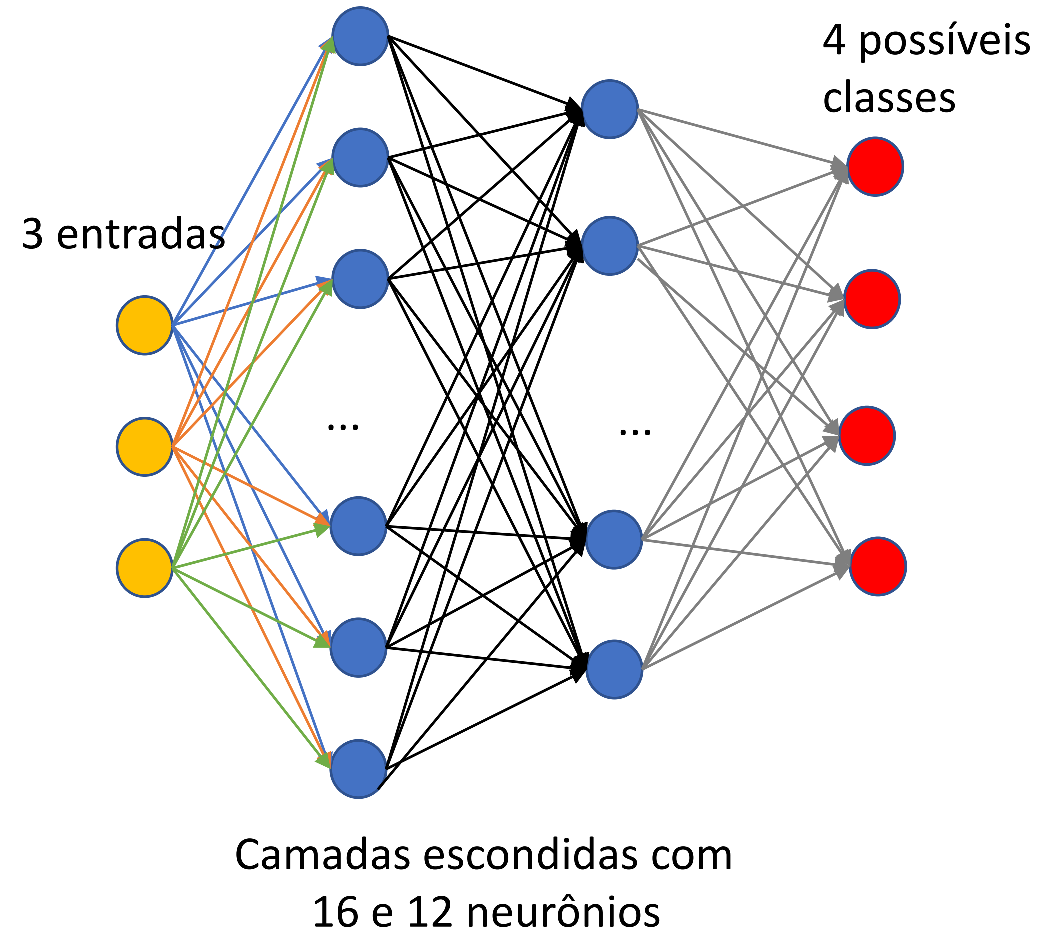
A saída é reformatada para ter 4 saídas, uma para cada classe. Quanto maior o valor produzido em cada neurónio maior a probabilidade do rio pertencer a essa classe.

0=0001

1=0010

2=0100

3=1000



Em Google Colab

```
# ler entradas: neste caso 3 variáveis que variam de 0-200
X=[[107, 117 ,115],
[219, 101 ,107],
[63, 106 ,55],
[114, 62 ,62],
[113, 101 ,107],
[209, 105 ,114],
…
# ler saídas: quatro classes 0,1,2,3
Y=[ [0],
[1],
[2],
[3],
[0],
...
```

Formatar entradas e saidas

```
Y=np.float16( Y )   # mudar saída para float
#Normaliar os dados de entrada:
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)

# mudar o formato da variável de saída para um arranjo binário com 4 el
ementos (0001, 0010, 0100, 1000)
# mostra qual neurônio é ativado na saída
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
y = ohe.fit_transform(Y).toarray()
```

calibracao e verificacao

```
# AGORA: separe em calibracao e verificacao
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.2)
print('os dados foram separados em train e test')
print(len(X_train) , len(X_test) )
```

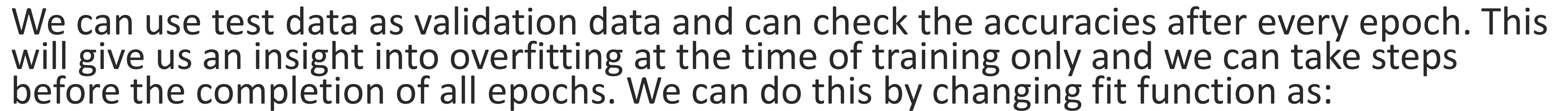
A rede

```
 importar bibliotecas de keras rede neural
import keras
from keras.models import Sequential
from keras.layers import Dense
model = Sequential()   # uma arranjo sequencial de camadas
model.add(Dense(16, input_dim=3, activation='relu')) # 1a camada, 16 neuronios
model.add(Dense(12, activation='relu'))              # 2a camada com 12
model.add(Dense(4, activation='softmax') )           # 3a camada com 4 saidas

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

```
#entao temos camadas de 16, 12 3 4 neuronios.
# na ultima camada, para fins ce classificação, usamos softmax como função de transferência.
# ou qual saída tem o maior valor?

# compilar a rede (montar a rede) definindo
# definimos loss "Categorical_crossentropy" especifica que temos várias classes.
# usado em casos em que uma amostra só pode pertencer a uma das classes e o modelo deve decidir qual delas.
# Formalmente, ele quantifica a diferença entre duas distribuições de probabilidade.
# optimizer = Adam. é uma variante do algoritmo de gradiente clássico usado nontreinamento (ADAptive Moment estimation):
# Metrics serve para definir como será avaliado o desempenho da rede: aqui avaliamos a acurácia da classifiacao
```

Treinar a rede

```
history = model.fit(
     X_train,
     y_train,
     epochs=100,
     batch_size=None
)
```



We can use test data as validation data and can check the accuracies after every epoch. This will give us an insight into overfitting at the time of training only and we can take steps before the completion of all epochs. We can do this by changing fit function as:

```
history = model.fit(
      X_train,
      y_train,
      validation_data = (X_test,y_test),
      epochs=100,
      validation_split=0.20,
      validation_split=0.20,
      batch_size=64
)
```

```
y_pred = model.predict(X_test)
# Converter a saida em numeros (labels)
pred = list()
for i in range(len(y_pred)):
    pred.append(np.argmax(y_pred[i]))
#Converting one hot encoded test label to label
test = list()
for i in range(len(y_test)):
    test.append(np.argmax(y_test[i]))
# calcular a acuracia final
from sklearn.metrics import accuracy_score
a = accuracy_score(pred,test)
print('Accuracy is:', a*100)
```

mostrar

```
print(history.history )
import matplotlib.pyplot as plt
plt.plot( history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test’],
loc='upper left')
plt.show()
```