

Aprendizagem Profunda com Matlab

Aprendizagem de Máquina, Redes Neurais e Inteligência Artificial

Professor Marco H. Terra

terra@sc.usp.br

Departamento de Engenharia Elétrica e de Computação
Universidade de São Paulo em São Carlos

Parte 2

September 15, 2020

MATLAB Deep Learning: With Machine Learning, Neural Networks and Artificial Intelligence

Phil Kim - Seoul, Soul-t'ukpyolsi, Korea (Republic of)

ISBN-13 (pbk): 978-1-4842-2844-9

ISBN-13 (electronic): 978-1-4842-2845-6

DOI 10.1007/978-1-4842-2845-6

Sumário

Rede Neural

Aprendizagem supervisionada

Treinamento de uma RN com uma camada: regra delta

Treinamento de uma RN com múltiplas camadas

Rede Neural

- ▶ Largamente utilizada como modelo para aprendizagem de máquina
- ▶ Grande interesse atual em rede neural (RN): aprendizagem profunda
- ▶ Como RN está relacionada com aprendizagem de máquina
- ▶ Aprendizagem de RN com uma única camada

Rede Neural

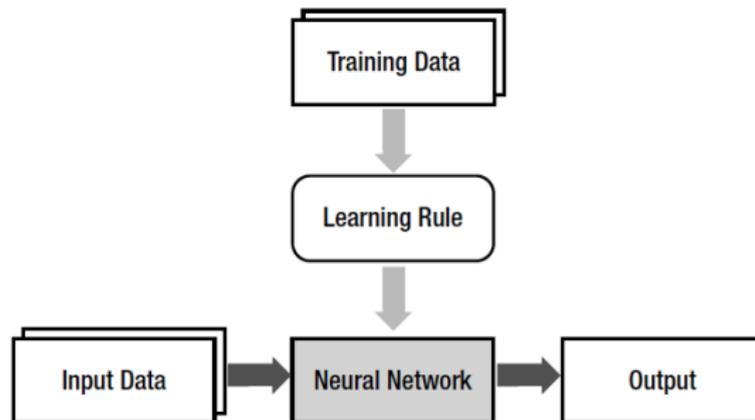


Figure 2: Relação entre aprendizagem de máquina e rede neural.

Nós de uma Rede Neural

- ▶ Neurônio não tem a capacidade de armazenar informação
- ▶ Apenas transmite sinais para outro neurônio
- ▶ O cérebro é uma rede gigante de neurônios
- ▶ A associação de neurônios forma uma informação específica
- ▶ A RN reproduz a associação dos neurônios, mecanismo mais importante do cérebro, usando os pesos

Nós de uma Rede Neural

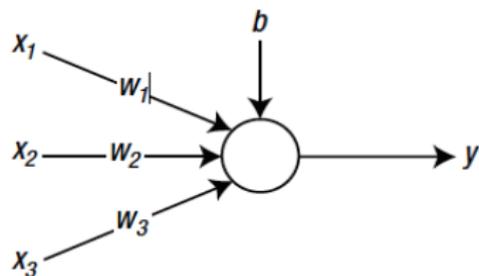


Figure 4: Um nó que recebe três entradas.

Nós de uma Rede Neural

- ▶ *Node of Ranvier, periodic gap in the insulating sheath (myelin) on the axon of certain neurons that serves to facilitate the rapid conduction of nerve impulses*
- ▶ Neurônio artificial (Figura 4): círculo e seta denotam o nó e o fluxo de sinal
- ▶ x_1 , x_2 e x_3 são os sinais de entrada
- ▶ w_1 , w_2 e w_3 são os pesos para os sinais correspondentes
- ▶ b é um fator de polarização

Nós de uma Rede Neural

- ▶ A soma ponderada do neurônio da Figura 4 é calculada da seguinte maneira

$$v = (w_1x_1) + (w_2x_2) + (w_3x_3) + b \quad (1)$$

$$v = wx + b \quad (2)$$

$$w = [w_1 \ w_2 \ w_3] \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (3)$$

- ▶ Este exemplo ilustra a função dos pesos na RN: imita como o cérebro altera a associação dos neurônios

Nós de uma Rede Neural

- ▶ Função de ativação: determina o comportamento dos nós

$$y = \phi(v) = \phi(wx + b) \quad (4)$$

Camadas da Rede Neural

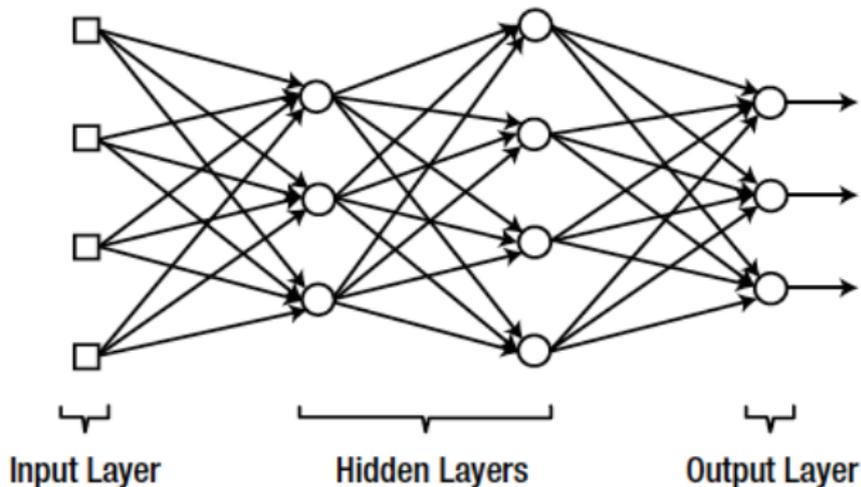


Figure 5: Uma estrutura por camadas dos nós.

Nós de uma Rede Neural

- ▶ RN com apenas camadas de entrada e de saída são chamadas de RN com camada simples
- ▶ Quando há uma camada escondida: *shallow or vanilla NN*
- ▶ Quando há duas ou mais camadas escondidas: RN profunda

Camadas da Rede Neural

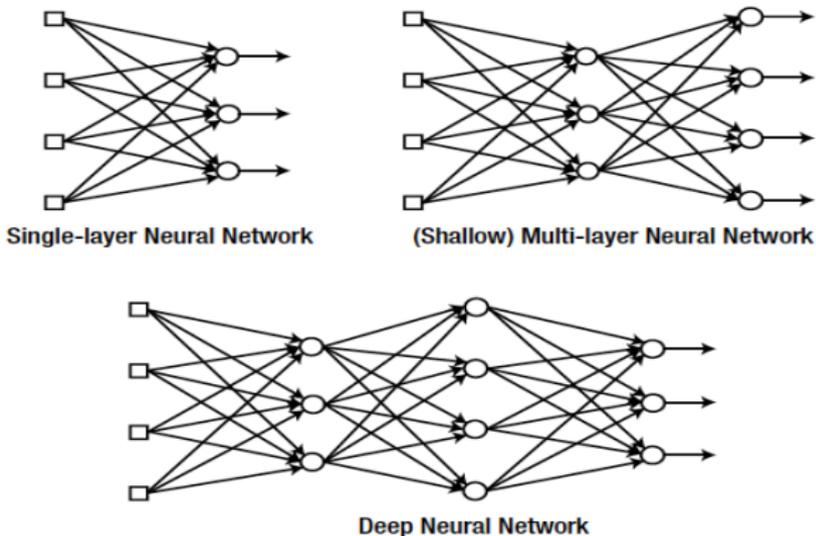
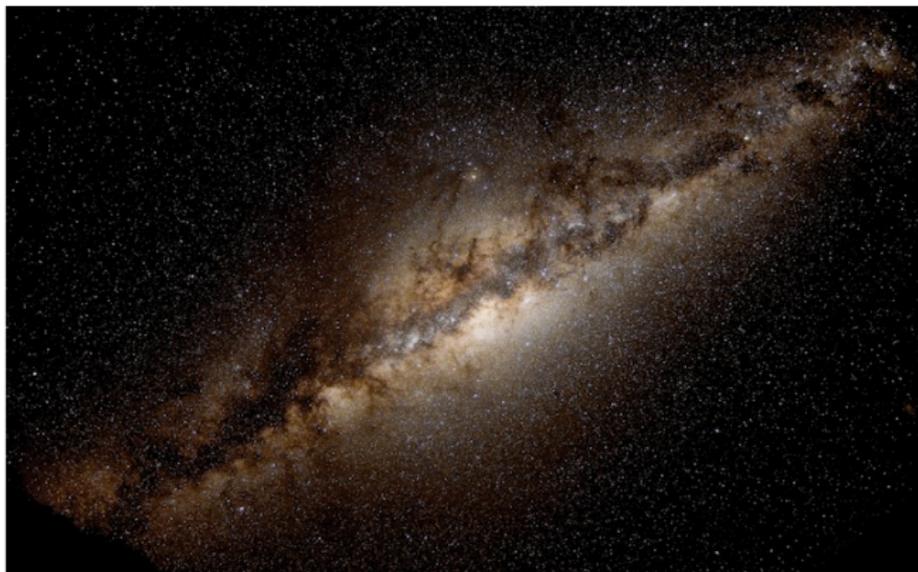


Figure 6: As definições de uma RN dependem da arquitetura das camadas.

Camadas da Rede Neural



(source: NASA Astronomy Picture of the Day)

Figure 7: Via Láctea.

Camadas da Rede Neural

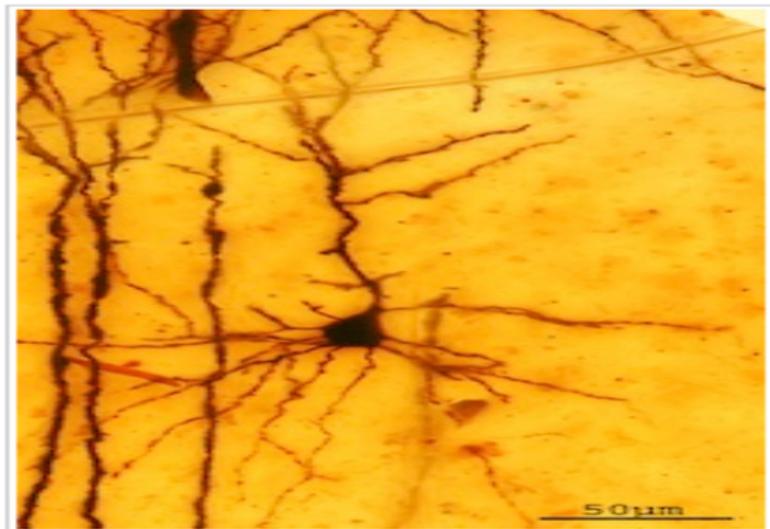


Figure 8: Neurônio.

Camadas da Rede Neural

- ▶ O cérebro é uma rede gigante de neurônios
- ▶ A RN é uma rede de nós
- ▶ Quantos neurônios, aproximadamente, temos no cérebro?

Camadas da Rede Neural

- ▶ O cérebro é uma rede gigante de neurônios
- ▶ A RN é uma rede de nós
- ▶ Quantos neurônios, aproximadamente, temos no cérebro?
- ▶ 86 bilhões de neurônios
- ▶ Quantas estrelas temos na Via Láctea?

Camadas da Rede Neural

- ▶ O cérebro é uma rede gigante de neurônios
- ▶ A RN é uma rede de nós
- ▶ Quantos neurônios, aproximadamente, temos no cérebro?
- ▶ 86 bilhões de neurônios
- ▶ Quantas estrelas temos na Via Láctea?
- ▶ Entre 200 e 400 bilhões

Camadas da Rede Neural

- Considere a seguinte RN com uma camada escondida:

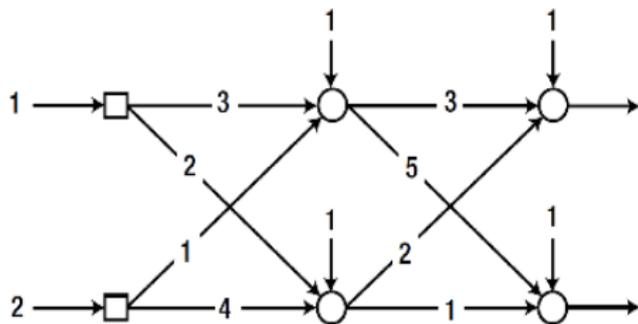


Figure 9: Rede neural com uma única camada escondida.

Camadas da Rede Neural

- ▶ Calcule as saídas da seguinte rede neural:

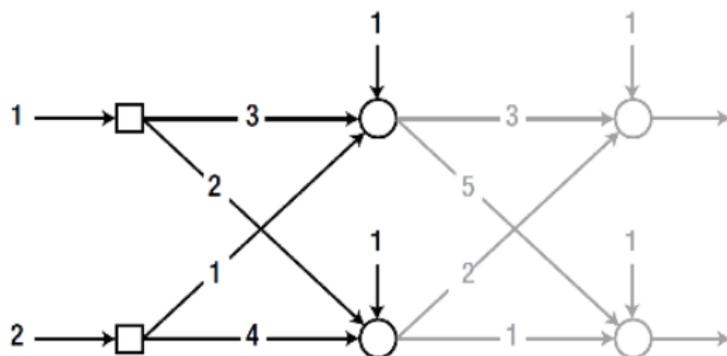


Figure 11: Cálculo da saída a partir da camada escondida.

Camadas da Rede Neural

- ▶ Soma ponderada, primeiro nó da camada escondida

$$\text{Soma ponderada: } v = (3 \times 1) + (1 \times 2) + 1 = 6 \quad (5)$$

$$\text{Saída: } y = \phi(v) = v = 6 \quad (6)$$

- ▶ Segundo nó da camada escondida

$$\text{Soma ponderada: } v = (2 \times 1) + (4 \times 2) + 1 = 11 \quad (7)$$

$$\text{Saída: } y = \phi(v) = v = 11 \quad (8)$$

Nós de uma Rede Neural

- ▶ A soma ponderada: relação matricial

$$v = \begin{bmatrix} 3x_1 + 1x_2 + 1 \\ 2x_1 + 4x_2 + 1 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 11 \end{bmatrix} \quad (9)$$

$$v = Wx + b \quad (10)$$



$$W = \begin{bmatrix} \text{pesos do primeiro nó} \\ \text{pesos do segundo nó} \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \quad (11)$$

Nós de uma Rede Neural

- ▶ Cálculo das saídas da próxima camada:

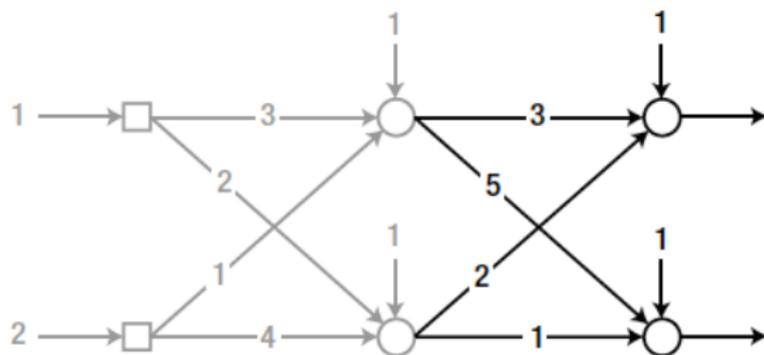


Figure 12: Camada oculta.

Nós de uma Rede Neural

- ▶ A soma ponderada:

$$v = \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 6 \\ 11 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 41 \\ 42 \end{bmatrix} \quad (12)$$

- ▶ Saída da camada oculta:

$$v = \phi(v) = v = \begin{bmatrix} 41 \\ 42 \end{bmatrix} \quad (13)$$

- ▶ Simplicidade algébrica para tratar problemas complexos

Aprendizagem de máquina

- Consequência de se adotar a função de ativação linear

$$v = \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 6 \\ 11 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (14)$$

$$= \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \left(\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (15)$$

$$= \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (16)$$

$$= \begin{bmatrix} 13 & 11 \\ 17 & 9 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 6 \\ 7 \end{bmatrix} \quad (17)$$

Aprendizagem de máquina

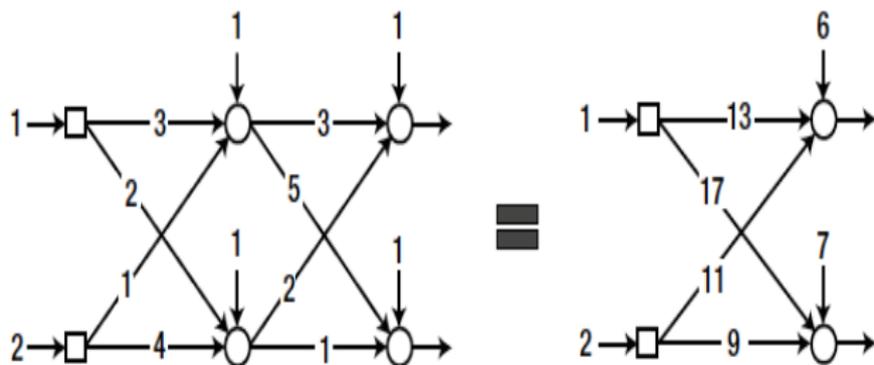


Figure 13: Equivalente a uma RN sem camada escondida.

Aprendizagem supervisionada

Procedimento para treinar uma RN supervisionada

- 1 Inicialize os pesos com valores adequados
- 2 Considere a entrada a partir dos dados de treinamento entrada, saída desejada. Insira esses dados na RN e calcule o erro da saída desejada
- 3 Ajuste os pesos para diminuir o erro
- 4 Repita os passos 2 e 3 para todos os dados de treinamento

Aprendizagem supervisionada

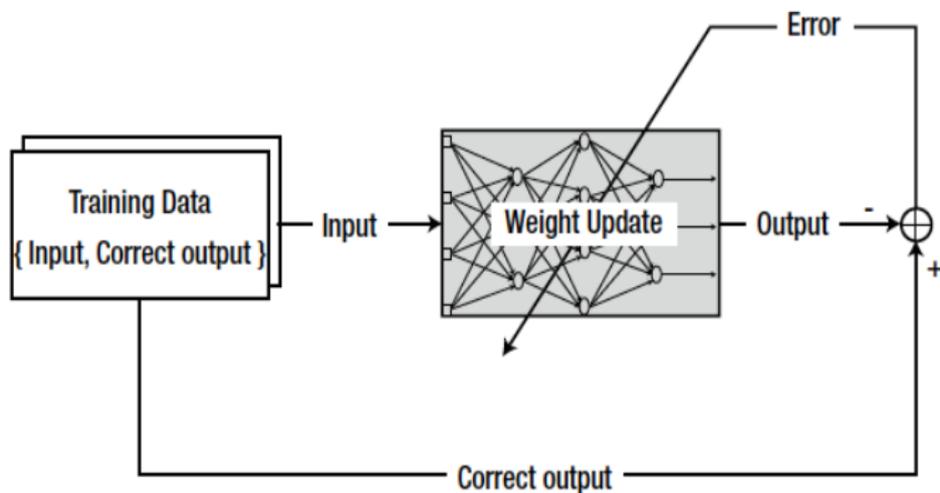


Figure 14: O conceito de rede supervisionada.

Treinamento de uma RN com uma camada

- ▶ Regra de aprendizagem é vital na pesquisa de RN
- ▶ Regra Delta: regra de aprendizagem representativa de uma RN com uma camada

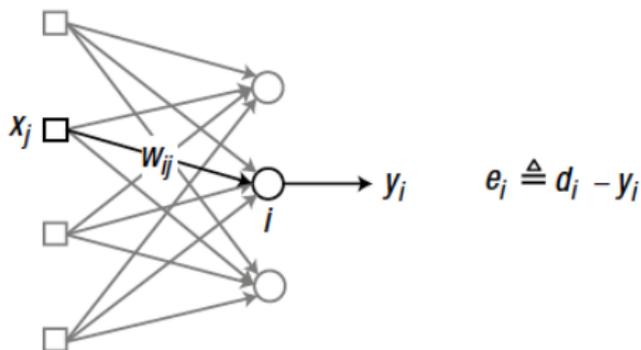


Figure 15: RN com uma camada

Treinamento de uma RN com uma camada

"Se um nó de entrada contribue para o erro do nó de saída, o peso entre os dois nós é ajustado em proporção ao valor da entrada, x_j e o erro de saída, e_i ."

$$w_{ij} \leftarrow w_{ij} + \alpha e_i x_j \quad (18)$$

- ▶ x_j = saída do nó de entrada j , ($j=1,2,3$)
- ▶ e_i = erro do nó de saída i
- ▶ w_{ij} = peso entre o nó de saída i e o nó de entrada j
- ▶ α = taxa de aprendizagem ($0 < \alpha \leq 1$)
(baixa, aprendizagem lenta / alta, aprendizagem rápida)

Exemplo

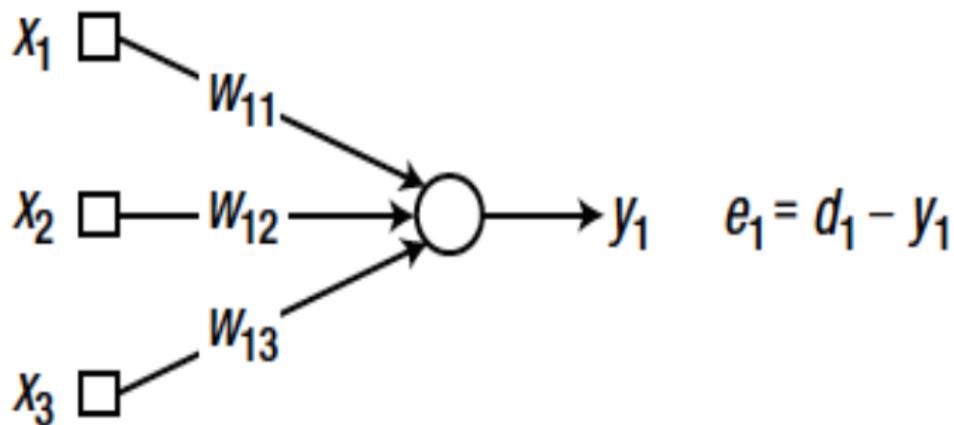


Figure 16: Rede neural com uma única camada com três nós de entrada e um nó de saída.

Exemplo

Regra delta

$$w_{11} \leftarrow w_{11} + \alpha e_1 x_1 \quad (19)$$

$$w_{12} \leftarrow w_{12} + \alpha e_1 x_2 \quad (20)$$

$$w_{13} \leftarrow w_{13} + \alpha e_1 x_3 \quad (21)$$

Resumo do processo de treinamento usando a regra delta para uma rede com uma única camada

- ▶ 1. Inicialize os pesos com valores adequados
- ▶ 2. Considere x_i , d_i e calcule

$$e_i = d_i - y_i$$

- ▶ 3. Calcule as taxas de ajuste dos pesos

$$\Delta w_{ij} = \alpha e_i x_j$$

- ▶ 4. Ajuste os peso com

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \quad (22)$$

- ▶ 5. Execute os passos 2-4 para todos os dados de treinamento
- ▶ 6. Repita os passos 2-5 até que o erro alcance um nível de tolerância aceitável

Gradiente descendente

- ▶ Época: número de iterações de treinamento
- ▶ Exemplo: época = n significa que a rede neural repete o processo de treinamento n vezes com o mesmo conjunto de dados
- ▶ A regra delta é o método do gradiente descendente

Regra delta generalizada

$$w_{ij} \leftarrow w_{ij} + \alpha_i \delta_i x_j \quad (23)$$

$$\delta_i = \dot{\phi}(v_i) e_i \quad (24)$$

- ▶ e_i = erro do nó de saída i
- ▶ v_i = soma ponderada do nó de saída i
- ▶ $\dot{\phi}$ = derivada da função de ativação ϕ do nó de saída i

Gradiente descendente

- ▶ Considerando a função de ativação linear $\phi(x) = x$, $\dot{\phi}(x) = 1$, então

$$\delta_i = e_i$$

- ▶ Função de ativação sigmoidal

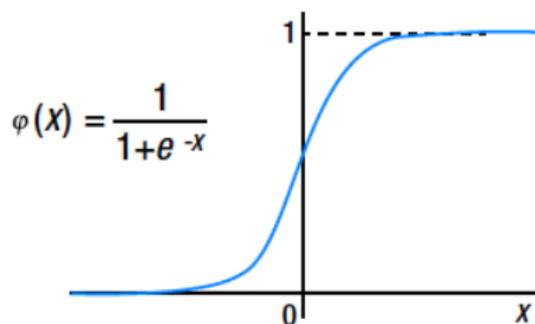


Figure 17: Função sigmoidal

Gradiente descendente

- ▶ Derivada da função sigmoidal

$$\dot{\phi}(x) = \frac{0 * (1 + e^{-x}) - (0 - e^{-x})}{(1 + e^{-x})^2} \quad (25)$$

$$= \frac{1}{(1 + e^{-x})} \times \left(1 - \frac{1}{(1 + e^{-x})} \right) \quad (26)$$

$$= \phi(x)(1 - \phi(x)) \quad (27)$$

- ▶ Regra delta e ajustes dos pesos

$$\delta_i = \dot{\phi}_i(v_i)e_i = \phi(v_i)(1 - \phi(v_i))e_i$$

$$w_{ij} \leftarrow w_{ij} + \phi(v_i)(1 - \phi(v_i))e_i x_j$$

Método Gradiente Descendente Estocástico (GDE)

- ▶ Calcula o erro para cada dado de treinamento e ajusta os pesos imediatamente
- ▶ Se tivermos n pontos de dados de treinamento o GDE ajusta os pesos n vezes

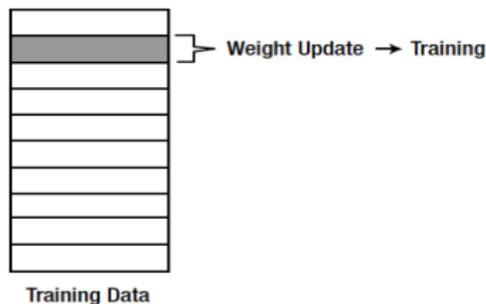


Figure 18: Como o ajuste dos pesos do GDE está relacionado com o conjunto dos dados de treinamento

Método Gradiente Descendente Estocástico (GDE)

- ▶ O GDE calcula os ajustes dos pesos com

$$\Delta w_{ij} = \alpha \delta_i x_j$$

Método *batch*

- ▶ No método *batch*, cada ajuste de peso é calculado para todos os erros dos dados de treinamento
- ▶ A média dos ajustes dos pesos é usada para ajustar os pesos



Figure 19: O método *batch* calcula o ajuste dos pesos e processo de treinamento.

Método *batch*

- ▶ Utiliza todos dados de treinamento e ajusta os pesos uma única vez
- ▶ O cálculo dos pesos é feito da seguinte maneira:

$$\Delta w_{ij} = \frac{1}{N} \sum_{k=1}^N \Delta w_{ij}(k)$$

sendo $\Delta_{ij}(k)$ é o ajuste do peso para o k -ésimo dado de treinamento e n é o número total dos dados de treinamento

- ▶ Cálculo da média dos ajustes dos pesos: maior consumo de tempo

Método *batch*

- ▶ Utiliza todos dados de treinamento e ajusta os pesos uma única vez
- ▶ O cálculo dos pesos é feito da seguinte maneira:

$$\Delta w_{ij} = \frac{1}{N} \sum_{k=1}^N \Delta w_{ij}(k)$$

sendo $\Delta_{ij}(k)$ é o ajuste do peso para o k -ésimo dado de treinamento e n é o número total dos dados de treinamento

- ▶ Cálculo da média dos ajustes dos pesos: maior consumo de tempo

Exemplo 1

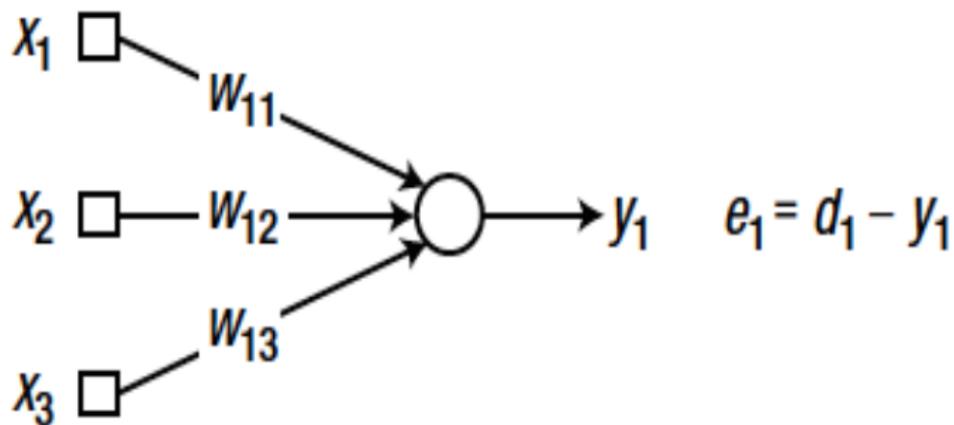


Figure 20: Rede neural com uma única camada com três nós de entrada e um nó de saída.

Exemplo 1

x_1	x_2	x_3	d_i
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

- ▶ Regra delta - função sigmoidal

$$\begin{aligned}\delta_i &= \phi(v_i)(1 - \phi(v_i))e_i \\ \Delta w_{ij} &= \alpha \delta_i x_j \\ w_{ij} &\leftarrow w_{ij} + \Delta w_{ij}\end{aligned}$$

- ▶ Aplicação dos métodos GDE e *batch*

Exemplo 1

Método do Gradiente Descendente Estocástico

- This program calls in the function DeltaBatch.m for supervised training of a neural network.

```
clear all clc
```

```
X = [0 0 1; 0 1 1; 1 0 1; 1 1 1]; D = [0; 0; 1; 1];
```

```
W = 2*rand(1, 3) - 1;
```

- Training (adjusting weights):

```
for epoch = 1:10000
```

```
W = DeltaBatch(W, X, D);
```

```
end
```

- Inference:

```
N = 4;
```

```
y = zeros(N,1);
```

```
for k = 1:N
```

```
x = X(k, :)';
```

```
v = W*x;
```

```
y(k) = Sigmoid(v); - obtained  
output.
```

```
end
```

```
disp('Results:'); disp(' [desired neuronoutput]'); disp([Dy]);
```

```
function W = DeltaBatch(W, X, D)
```

```
alpha = 0.9;
```

```
dWsum = zeros(3, 1);
```

```
N = 4;
```

```
for k = 1:N
```

```
    x = X(k, :);
```

```
    d = D(k);
```

```
    v = W * x;
```

```
    y = Sigmoid(v);
```

```
    e = d - y;
```

```
    delta = y * (1 - y) * e;
```

```
    dW = alpha * delta * x;
```

```
    dWsum = dWsum + dW;
```

```
end
```

```
dWavg = dWsum/N;
```

```
W(1) = W(1) + dWavg(1);
```

```
W(2) = W(2) + dWavg(2);
```

```
W(3) = W(3) + dWavg(3);
```

```
end
```

```
function y = Sigmoid(x)
y = 1 / (1 + exp(-x));
end
```

Resultado:

saída real	d_i
0.0102	0
0.0083	0
0.9932	1
0.9917	1

w_1	w_2	w_3
9.5649	-02084	-4.5752

Exemplo 1

Método *Batch*

Exemplo 1

- This program calls in the function DeltaBatch.m for supervised training of a neural network.

```
clear all; clc
```

```
X = [0 0 1; 0 1 1; 1 0 1; 1 1 1]; D = [0; 0; 1; 1];
```

```
W = 2*rand(1, 3) - 1;
```

```
for epoch = 1:40000
```

```
W = DeltaBatch(W, X, D);
```

```
end
```

- Inference:

```
N = 4;
```

```
y = zeros(N,1);
```

```
for k = 1:N
```

```
x = X(k, :);
```

```
v = W*x;
```

```
y(k) = Sigmoid(v);
```

```
end
```

```
disp('Results:');
```

```
disp('[desired neuron-output]');
```

```
disp([D y]);
```

Exemplo 1

```
function W = DeltaBatch(W, X, D);  
alpha = 0.9;  
dWsum = zeros(3, 1);  
N = 4;  
for k = 1:N  
    x = X(k, :)';  
    d = D(k);  
    v = W*x;  
    y = Sigmoid(v);  
    e = d - y;  
    delta = y*(1-y)*e;  
    dW = alpha*delta*x;  
    dWsum = dWsum + dW;  
end  
dWavg = dWsum/N;  
W(1) = W(1) + dWavg(1);  
W(2) = W(2) + dWavg(2);  
W(3) = W(3) + dWavg(3);  
end
```

Resultado:

saída real	d_i
0.0102	0
0.0083	0
0.9932	1
0.9917	1

w_1	w_2	w_3
9.5642	-02088	-4.5746

Comparação entre GDE e *Batch* Erros médios

Exemplo 1

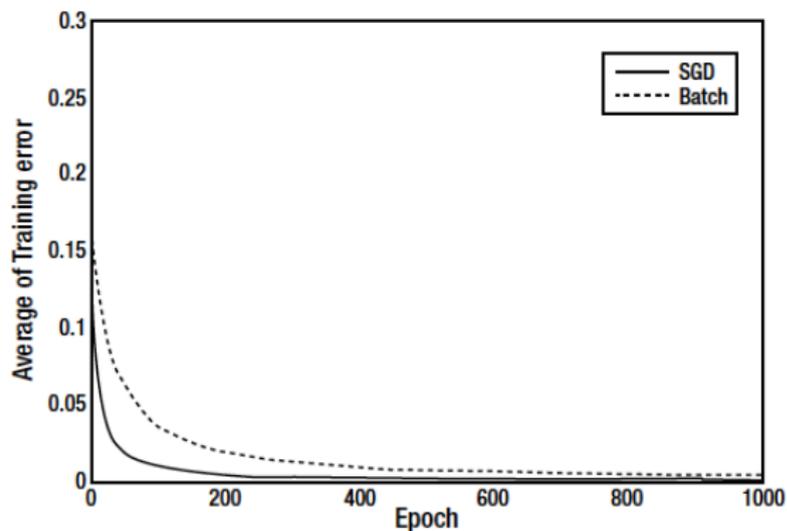


Figure 21: O método Gradiente Descendente Estocástico aprende mais rápido que o método *Batch*.

Exemplo 1

- The weights of both methods are initialized with the same values.

```
X = [0 0 1;  
0 1 1;  
1 0 1;  
1 1 1];  
D = [0; 0; 1; 1];  
E1 = zeros(1000, 1);  
E2 = zeros(1000, 1);  
W1 = 2*rand(1, 3) - 1;  
W2 = W1;  
    for epoch = 1:1000  
        W1 = DeltaSGD(W1, X, D);  
        W2 = DeltaBatch(W2, X, D);  
        es1 = 0;  
        es2 = 0;  
        N = 4;  
        for k = 1:N  
            x = X(k, :);  
            d = D(k);  
            v1 = W1*x;  
            y1 = Sigmoid(v1);  
            es1 = es1 + (d - y1)^2;  
            v2 = W2*x;  
            y2 = Sigmoid(v2);  
            es2 = es2 + (d - y2)^2;  
        end  
        E1(epoch) = es1/N;  
        E2(epoch) = es2/N;  
    end
```

```
plot(E1, 'r', 'LineWidth', 1.5)
hold on
plot(E2, 'b:', 'LineWidth', 1.5)
xlabel('Epoch')
ylabel('Average of Training error')
legend('SGD', 'Batch')
```

Limitações das RNs com camadas simples

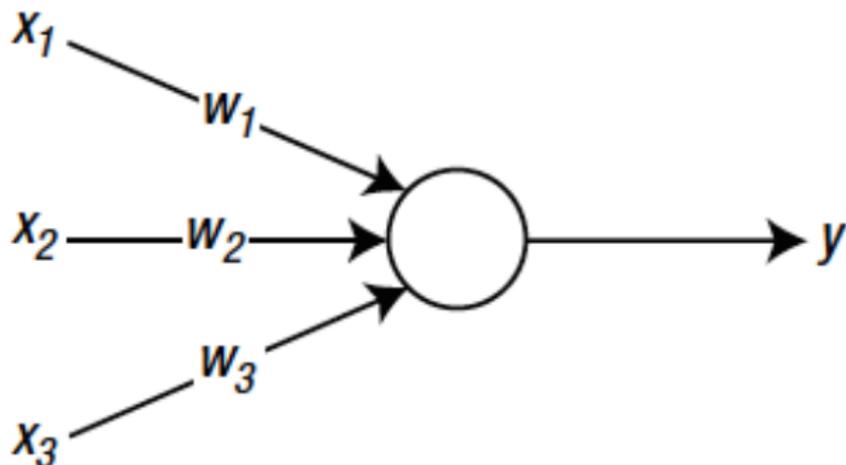


Figure 22: Rede neural com uma única camada.

Limitações das RNs com camadas simples

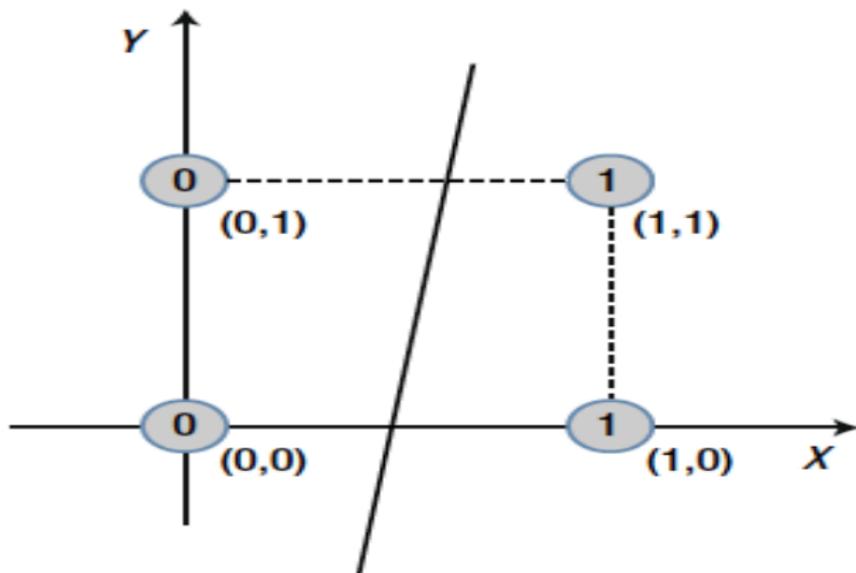


Figure 26: Problema de dados linearmente separáveis. Se não for separável, a RN necessita mais camadas.

Treinamento de uma RN com múltiplas camadas

RN com múltiplas camadas

- ▶ Consegue classificar classes não linearmente separáveis (limitação da RN com uma única camada);
- ▶ Presença de camadas escondidas (intermediárias):
 - ▶ Neurônios escondidos extraem a maior parte das informações contidas nas amostras;
- ▶ Principais aplicações: **classificador**, aproximador de funções, otimização, previsão de séries temporais, identificação e controle de processos;
- ▶ Treinamento supervisionado: algoritmo **back-propagation**.

Algoritmo back-propagation

- ▶ A regra Delta sozinha não consegue treinar uma RN multicamadas:
 - ▶ Não são definidos erros para camadas escondidas;
- ▶ O algoritmo back-propagation define os erros para as camadas escondidas:
 - ▶ Uma vez definidos esses erros, a regra Delta pode ser usada para ajustar os pesos em cada camada;
- ▶ Os sinais de entrada e erro de saída fluem em sentidos contrários:
 - ▶ Sinais de entrada → sentido direto (*forward propagation*);
 - ▶ Sinais de erro de saída → sentido reverso (*backward propagation*).

Algoritmo back-propagation

Considere a seguinte RN multicamadas com dois neurônios na camada escondida e dois neurônios na camada de saída:

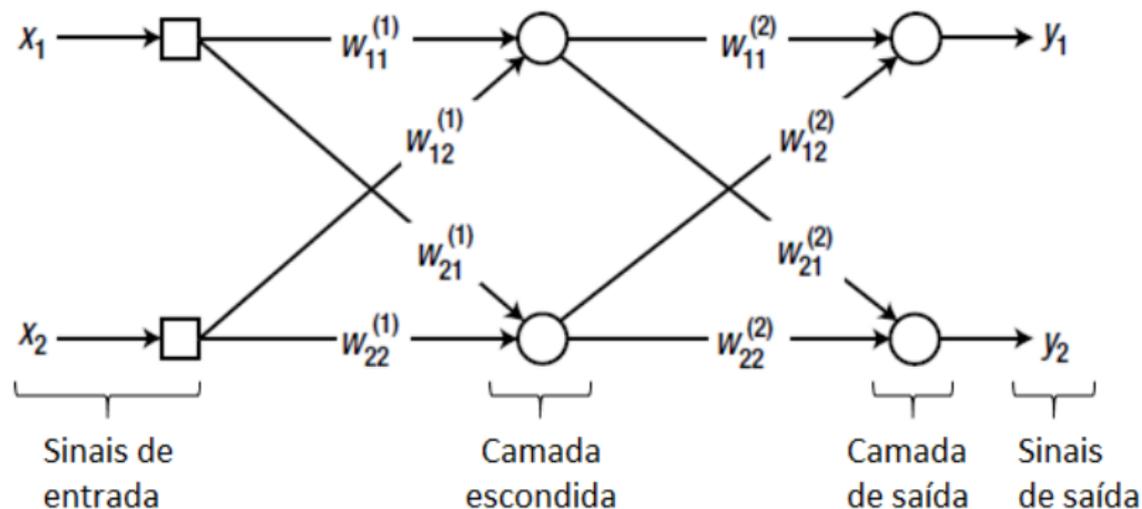


Figure 27: RN multicamadas com uma camada escondida. Fonte: [1].

Algoritmo back-propagation

1º: propagar o sinal de entrada $[x_1 \ x_2]^T$ pela camada intermediária.

$$\begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = W_1 x,$$

e então:

$$\begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} = \begin{bmatrix} \varphi \left(v_1^{(1)} \right) \\ \varphi \left(v_2^{(1)} \right) \end{bmatrix},$$

onde $\varphi(\cdot)$ é a função de ativação, $y_i^{(k)}$ é a saída do i -ésimo neurônio da k -ésima camada, $w_{ji}^{(k)}$ é o peso que conecta o j -ésimo neurônio da k -ésima camada neural ao i -ésimo neurônio da camada anterior. (Perceba que se $k = 1$, $w_{ji}^{(1)}$ conecta os neurônios da 1ª camada escondida aos sinais de entrada.)

Algoritmo back-propagation

2º: propagar as saídas da camada escondida para obter os sinais de saída da rede.

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \end{bmatrix} \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} = W_2 y^{(1)},$$

e então:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \varphi(v_1) \\ \varphi(v_2) \end{bmatrix},$$

onde y_i é a saída do i -ésimo neurônio da camada de saída.

Algoritmo back-propagation

3º: calcular os erros e δ_i (o mesmo da regra Delta) da camada de saída.

$$e_1 = d_1 - y_1,$$

$$e_2 = d_2 - y_2,$$

$$\delta_1 = \varphi'(v_1)e_1,$$

$$\delta_2 = \varphi'(v_2)e_2,$$

onde e_i é o erro obtido pelo i -ésimo neurônio da camada de saída, d_i é o valor desejado e $\varphi'(\cdot)$ é a derivada da função de ativação.

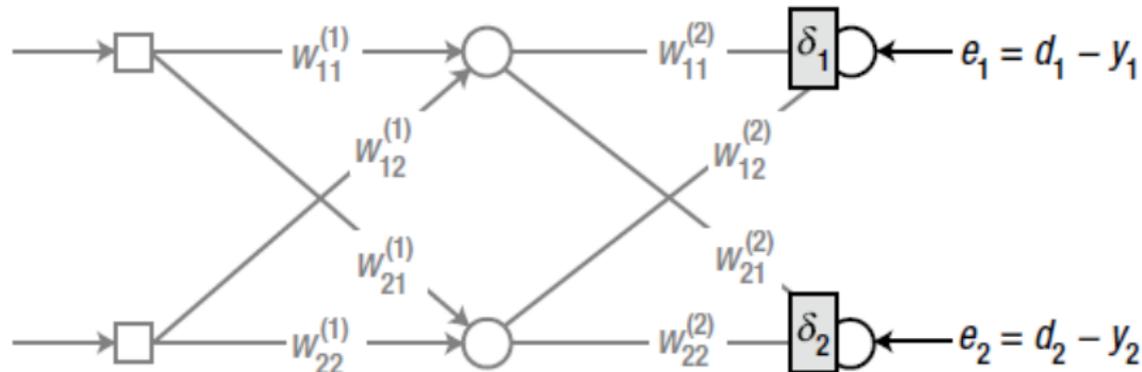


Figure 28: RN multicamadas com os δ s do back-propagation. Fonte: [1].

Algoritmo back-propagation

Perceba que $e_1^{(1)}$ e $e_2^{(1)}$ podem ser calculados por:

$$\begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} = W_2^T \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} \Rightarrow \text{útil para implementação.}$$

Obs.: se existirem mais camadas escondidas com diferentes números de neurônios, o mesmo processo se repete para cada uma delas e os respectivos deltas são calculados.

Algoritmo back-propagation

5º: ajustar os pesos sinápticos da k-ésima camada neural.

$$\Delta w_{ji}^{(k)} = \alpha \delta_i^{(k)} x_{ji},$$

$$w_{ji}^{(k)} \leftarrow w_{ji}^{(k)} + \Delta w_{ji}^{(k)},$$

onde x_{ji} é o sinal que multiplica o peso $w_{ji}^{(k)}$ e α é a taxa de aprendizagem tal que $\alpha \in (0, 1]$.

Algoritmo back-propagation

Exemplo: ajuste dos pesos $w_{21}^{(2)}$ e $w_{11}^{(1)}$.

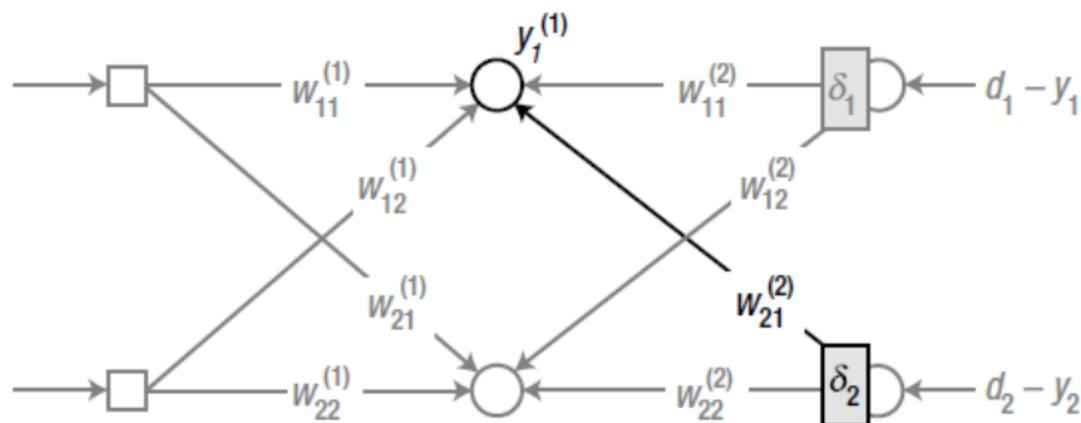


Figure 30: Ajuste de $w_{21}^{(2)}$. Fonte: [1].

Ajuste:

$$\Delta w_{21}^{(2)} = \alpha \delta_2 y_1^{(1)}, \quad w_{21}^{(2)} \leftarrow w_{21}^{(2)} + \Delta w_{21}^{(2)},$$

onde $y_1^{(1)}$ é a saída do primeiro nó da camada escondida.

Algoritmo back-propagation

Continuação.

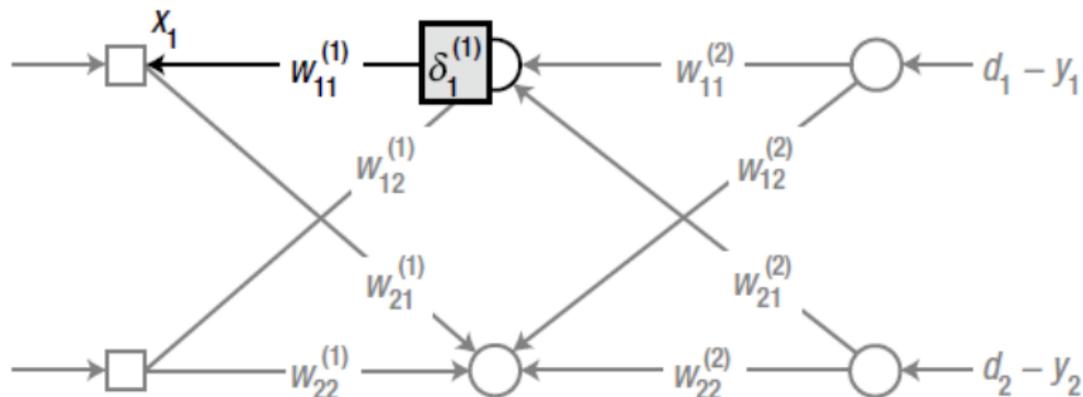


Figure 31: Ajuste de $w_{11}^{(1)}$. Fonte: [1].

Ajuste:

$$\Delta w_{11}^{(1)} = \alpha \delta_1^{(1)} x_1, \quad w_{11}^{(1)} \leftarrow w_{11}^{(1)} + \Delta w_{11}^{(1)},$$

onde x_1 é o primeiro sinal de entrada da rede neural.

Algoritmo back-propagation

Algoritmo:

1. Inicializar os pesos (geralmente de forma aleatória com $w_{ji}^{(k)} \in [-1, 1]$);
2. Apresentar os dados de entrada do conjunto de treinamento e obter a saída da rede. Calcular os erros de saída e os δ s dos neurônios de saída:

$$e = d - y, \quad \delta = \varphi'(v)e;$$

3. Propagar para trás (*backward*) os δ s e calcular os $\delta^{(k)}$ da camada imediatamente à esquerda:

$$e^{(k)} = W^T \delta, \quad \delta^{(k)} = \varphi'(v^{(k)}) e^{(k)};$$

4. Repetir o passo 3 até alcançar a primeira camada neural escondida (que fica imediatamente após a entrada da rede).

Algoritmo back-propagation

Algoritmo (continuação):

5. Ajustar o pesos segundo a seguinte regra de treinamento:

$$\Delta w_{ji} = \alpha \delta_i x_{ji},$$
$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

6. Repetir os passos 2-5 para cada amostra do conjunto de treinamento (completar 1 época);
7. Repetir os passos 2-6 até que seja atingida alguma das condições de parada (número máximo de épocas ou norma mínima desejada para o erro na saída).

Variações do back-propagation: termo de momentum

Inserção do termo de momentum (m): aumenta a velocidade do treinamento ao usar a regra delta.

O passo 5 do algoritmo é modificado da seguinte maneira:

$$\begin{aligned}\Delta w &= \alpha \delta x, \\ m &= \Delta w + \beta m^-, \\ w &\leftarrow w + m, \\ m^- &\leftarrow m,\end{aligned}$$

onde m^- é o termo de momentum da iteração anterior à atual e $0 < \beta < 1$.

Variações do back-propagation: termo de momentum

Vejamos como o momentum evolui ao longo das iterações:

$$m(0) = 0,$$

$$m(1) = \Delta w(1) + \beta m(0) = \Delta w(1),$$

$$m(2) = \Delta w(2) + \beta m(1) = \Delta w(2) + \beta \Delta w(1),$$

$$m(3) = \Delta w(3) + \beta m(2) = \Delta w(3) + \beta \Delta w(2) + \beta^2 \Delta w(1),$$

⋮

$$m(t) = \sum_{i=1}^t \beta^{t-i} \Delta w(i).$$

Note que inicialmente $m(t)$ tem grande contribuição no ajuste do peso (ajuste maior que apenas Δw). Ao longo das iterações na época, a contribuição de βm^- diminui. Logo, $m(t)$ faz o treinamento convergir mais rápido e mantém a estabilidade.

Variações do back-propagation: entropia cruzada

- ▶ Percebam que o back-propagation trabalha no sentido de diminuir o erro da saída \Rightarrow a soma dos erros quadráticos da rede é a função de custo usada na derivação do algoritmo.

$$J = \sum_{i=1}^M \frac{1}{2} (d_i - y_i)^2.$$

- ▶ Podemos adotar outra função de custo: entropia cruzada (*cross-entropy*):

$$J = \sum_{i=1}^M [-d_i \ln(y_i) - (1 - d_i) \ln(1 - y_i)].$$

Variações do back-propagation: entropia cruzada

- ▶ A função de entropia cruzada apresenta os termos $\ln(y_i)$ e $\ln(1 - y_i)$. Logo,
 - ▶ é necessário que $0 < y_i < 1$;
 - ▶ geralmente são usadas funções sigmóide ou *softmax* para ativação:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{k=1}^M x_k}, \quad x \in \mathbb{R}^k.$$

- ▶ O uso de logaritmos faz com que a entropia cruzada seja mais sensível aos erros de saída do que a soma dos erros quadráticos.
 - ▶ Entropia cruzada apresenta melhor performance;
 - ▶ Sempre que permitido pelas características da aplicação, deve-se escolher regras de aprendizagem baseadas em entropia cruzada.

Variações do back-propagation: entropia cruzada

Algoritmo back-propagation com função de custo de entropia cruzada:

1. Inicializar os pesos (geralmente de forma aleatória com $w_{ji}^{(k)} \in [-1, 1]$);
2. Apresentar os dados de entrada do conjunto de treinamento e obter a saída da rede. Calcular os erros de saída e os δ s dos neurônios de saída:

$$e = d - y, \quad \delta = e;$$

3. Propagar para trás (*backward*) os δ s e calcular os $\delta^{(k)}$ da camada imediatamente à esquerda:

$$e^{(k)} = W^T \delta, \quad \delta^{(k)} = \varphi' \left(v^{(k)} \right) e^{(k)};$$

4. Repetir o passo 3 até alcançar a primeira camada neural escondida (que fica imediatamente após a entrada da rede).

Variações do back-propagation: entropia cruzada

Algoritmo back-propagation com função de custo de entropia cruzada (continuação):

5. Ajustar o pesos segundo a seguinte regra de treinamento:

$$\begin{aligned}\Delta w_{ji} &= \alpha \delta_i x_{ji}, \\ w_{ji} &\leftarrow w_{ji} + \Delta w_{ji}\end{aligned}$$

6. Repetir os passos 2-5 para cada amostra do conjunto de treinamento (completar 1 época);
7. Repetir os passos 2-6 até que seja atingida alguma das condições de parada (número máximo de épocas ou norma mínima desejada para o erro na saída).

Perceba que a única diferença entre esse algoritmo e o do back-propagation com soma de erros quadráticos está no cálculo de δ no passo 2.

Algoritmo back-propagation: implementação em Matlab

Código 1: BackpropXOR.m.

```
function [W1, W2] = BackpropXOR(W1, W2, X, D)
```

```
% This function performs backpropagation SGD training of a neural  
network with one hidden layer.
```

```
% W1: weight matrix between the input layer and hidden layer.
```

```
% W2: weight matrix between the hidden layer and output layer.
```

```
% X: inputs for supervised training.
```

```
% D: desired outputs.
```

```
alpha = 0.9;
```

```
N = length(X);
```

```
for k = 1:N
```

```
    x = X(k, :);
```

```
    d = D(k);
```

```
    v1 = W1*x;
```

```
    y1 = Sigmoid(v1);
```

Algoritmo back-propagation: implementação em Matlab

Código 1: BackpropXOR.m (continuação).

```
v = W2*y1;  
y = Sigmoid(v);  
e = d - y;  
  
delta = y.*(1-y).*e;  
e1 = W2'*delta;  
delta1 = y1.*(1-y1).*e1;  
dW1 = alpha*delta1*x';  
W1 = W1 + dW1;  
  
dW2 = alpha*delta*y1';  
W2 = W2 + dW2;
```

```
end %for  
end %function
```

Algoritmo back-propagation: implementação em Matlab

Código 2: TestBackpropXOR.m.

% This program calls in the BackpropXOR.m function and trains the neural network max_epoch times.

```
clear all  
clc
```

%Inputs:

```
X = [0 0 1; 0 1 1; 1 0 1; 1 1 1];
```

%Desired outputs:

```
D = [0; 1; 1; 0];
```

%Initialization of weights:

```
W1 = 2*rand(4, 3) - 1;
```

```
W2 = 2*rand(1, 4) - 1;
```

% Training process (adjusting weights):

```
max_epoch = 10000;
```

```
for epoch = 1:max_epoch      %train
```

```
    [W1, W2] = BackpropXOR(W1, W2, X, D);
```

```
end
```

Algoritmo back-propagation: implementação em Matlab

Código 2: TestBackpropXOR.m (continuação).

```
%Inference:
N = size(X,1);
y = zeros(N,1);
for k = 1:N
    x = X(k, :)' ;
    v1 = W1*x;
    y1 = Sigmoid(v1);
    v = W2*y1;
    y(k) = Sigmoid(v); %obtained output.
end

disp('Results:');
disp(' [desired      neuron_output]');
disp([D y]);
```

Algoritmo back-propagation: implementação em Matlab

Código 3: BackpropCE.m.

```
function [W1, W2] = BackpropCE(W1, W2, X, D)
```

```
%Backpropagation with Cross Entropy cost function.
```

```
alpha = 0.9;
```

```
N = size(X,1);
```

```
for k = 1:N
```

```
    x = X(k, :)' ; %input x is a column vector
```

```
    d = D(k); %desired output
```

```
    v1 = W1*x;
```

```
    y1 = Sigmoid(v1);
```

```
    v = W2*y1;
```

```
    y = Sigmoid(v); %out. neuron in output layer.
```

```
    e = d - y;
```

```
    delta = e;
```

Algoritmo back-propagation: implementação em Matlab

Código 3: BackpropCE.m (continuação).

```
e1 = W2'*delta;  
delta1 = y1.*(1-y1).*e1;  
dW1 = alpha*delta1*x';  
W1 = W1 + dW1;  
dW2 = alpha*delta*y1';  
W2 = W2 + dW2;  
  
end %for  
end %function
```

Algoritmo back-propagation: implementação em Matlab

Código 4: TestBackpropCE.m.

```
% This program calls the BackpropCE.m function and trains the neural network max_epoch times.
```

```
clear all  
clc
```

```
%Inputs:
```

```
X = [0 0 1; 0 1 1; 1 0 1; 1 1 1];
```

```
%Desired outputs:
```

```
D = [0; 1; 1; 0];
```

```
%Initialization of weights:
```

```
W1 = 2*rand(4, 3) - 1;
```

```
W2 = 2*rand(1, 4) - 1;
```

```
% Training process (adjusting weights):
```

```
max_epoch = 10000;
```

```
for epoch = 1:max_epoch      %train
```

```
    [W1, W2] = BackpropCE(W1, W2, X, D);
```

```
end
```

Algoritmo back-propagation: implementação em Matlab

Código 4: TestBackpropCE.m (continuação).

```
%Inference:
N = size(X,1);
y = zeros(N,1);
for k = 1:N
    x = X(k, :)' ;
    v1 = W1*x;
    y1 = Sigmoid(v1);
    v = W2*y1;
    y(k) = Sigmoid(v); %obtained output.
end

disp('Results:');
disp(' [desired      neuron_output]');
disp([D y]);
```

Algoritmo back-propagation: exemplo 1

Executando o código 2 (TestBackpropXOR.m) para realizar a operação XOR. RN com 3 entradas, 1 camada escondida com 4 neurônios e 1 neurônio na camada de saída.

saída real	d_i
0.0106	0
0.9900	1
0.9902	1
0.0110	0

$$W1 = \begin{bmatrix} -3.4272 & -3.6019 & 5.3359 \\ 3.1907 & 2.6050 & -0.0809 \\ -6.8773 & -6.6293 & 2.9185 \\ 3.3357 & 3.1405 & -4.9207 \end{bmatrix},$$

$$W2 = \begin{bmatrix} 6.1638 \\ 0.7844 \\ -11.5907 \\ -7.5517 \end{bmatrix}^T$$

Algoritmo back-propagation: exemplo 2

Executando o código 4 (TestBackpropCE.m) para realizar a operação XOR. RN com 3 entradas, 1 camada escondida com 4 neurônios e 1 neurônio na camada de saída.

saída real	d_i
0.0000	0
0.9999	1
0.9998	1
0.0004	0

$$W1 = \begin{bmatrix} 4.7913 & -3.0530 & 1.7926 \\ -8.8875 & 4.2089 & -1.1488 \\ -6.6164 & -7.4783 & 1.9195 \\ 6.5946 & -9.8156 & -2.2504 \end{bmatrix}, \quad W2 = \begin{bmatrix} -8.2394 \\ 11.7559 \\ -9.1725 \\ 16.8448 \end{bmatrix}^T$$

Algoritmo back-propagation: exemplo 3

Convergência da treinamento com diferentes funções de custo: soma dos erros quadráticos (*sum of squared errors*) e entropia cruzada (*cross entropy*).¹ RN com 3 entradas, 1 camada escondida com 4 neurônios e 1 neurônio na camada de saída.

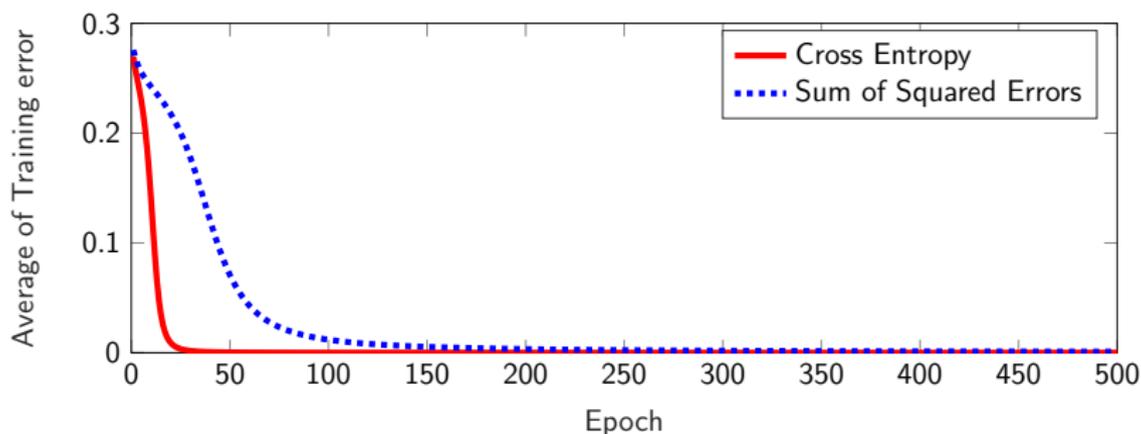


Figure 32: Convergência do treinamento com soma dos erros quadráticos e entropia cruzada.

¹Veja o código CEvsSSE.m no repositório da disciplina.