Introdução às Redes Neurais Artificiais com Implementações em R

Ricardo Cerri[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Rodovia Washington Luís, km 235, Jardim Guanabara
13565-905 – São Carlos – SP – Brasil

cerri@ufscar.br

Abstract. *Artificial Neural Networks are computer systems based on the human brain. They are formed by connected artificial neurons that share information for solving problems in various applications, such as pattern recognition, security systems, medicine, and autonomous cars. Their variants range from simple architectures to complex models with several layers of interconnected neurons. This short course presents fundamentals about three basic models of Artificial Neural Networks: Perceptron, Multi-Layer Perceptron, and Autoencoder.*

Resumo. *Redes Neurais Artificiais são sistemas computacionais baseados no funcionando do cérebro humano. Elas são formadas por neurônios artificiais conectados que compartilham informações para a solução de problemas em diversas aplicações, como o reconhecimento de padrões, sistemas de segurança, medicina, e carros autônomos. Suas variantes vão desde arquiteturas simples até modelos complexos com várias camadas de neurônios interconectados. Este minicurso apresenta fundamentações sobre três modelos básicos de Redes Neurais Artificiais: o Perceptron, o Multi-Layer Perceptron, e o Autoencoder.*

1. Introdução

As Redes Neurais Artificiais (RNAs) podem ser definidas como sistemas computacionais que processam informações baseados no funcionamento do cérebro e nas conexões de seus neurônios. A rede adquire conhecimento por meio de um processo de aprendizado, e as forças das conexões entre os neurônios artificiais da rede, conhecidas como pesos sinápticos, são utilizadas para armazenar o conhecimento adquirido [Haykin 1999].

A Figura 1 ilustra um neurônio artificial proposto por [McCulloch e Pitts 1943] com suas partes mais importantes: as sinapses, caracterizadas por pesos associados a cada entrada do neurônio, uma função somadora, e uma função de ativação. A função de ativação limita a amplitude do sinal de saída do neurônio a algum valor finito [Rumelhart e McClelland 1986]. Várias funções de ativação podem ser utilizadas, como a função linear e a função sigmoidal.

O bias b_k da Figura 1 tem o efeito de aumentar ou diminuir o valor de entrada da função de ativação, dependendo se esse valor for positivo ou negativo, respectivamente [Haykin 1999]. O neurônio da Figura 1 pode ser representado pela Equação 1.

$$y_k = f(u_k) = f\left(\sum_{j=1}^{m} w_{kj}x_j + b_k\right) \tag{1}$$

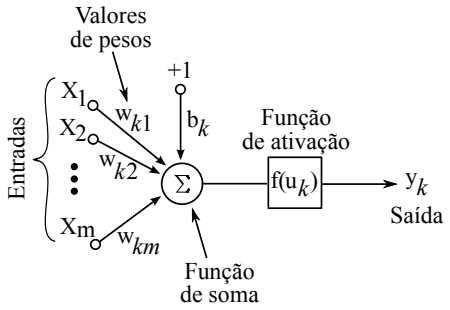

Figura 1. Ilustração de um Neurônio Artificial. Adaptado de [Haykin 1999].

2. O Perceptron

O *Perceptron* utiliza o modelo de neurônio proposto por McCulloch e Pitts. A adaptação dos pesos sinápticos do *Perceptron* acontece da seguinte maneira [Haykin 1999]:

1. **Inicialização:** inicializar os pesos ($\mathbf{w}$) aleatoriamente com valores pequenos;
2. **Apresentação dos exemplos de treinamento:** para cada exemplo de treinamento $\mathbf{x}_i$, executar os passos 3 e 4;
3. **Cálculo da resposta do neurônio:** a saída do neurônio para um exemplo de entrada $\mathbf{x}_i$ é calculada utilizando uma função de ativação $f(\cdot)$: $y_i = f(\mathbf{w} \times \mathbf{x}_i)$
4. **Atualização do vetor de pesos:** os pesos sinápticos são atualizados: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta[d_i - y_i]\mathbf{x}_i$
5. **Continuação:** Repetir passos 2, 3 e 4 até que um critério de parada seja satisfeito.

No processo de aprendizado, η representa a taxa de aprendizado do *Perceptron*, e $d_i - y_i$ representa o erro obtido pela diferença entre a saída do neurônio (y_i) e a saída desejada (d_i) do exemplo $\mathbf{x}_i$. Esses mesmos passos de aprendizado podem ser utilizados se for considerada uma camada formada por vários *perceptrons*, de forma a obter múltiplas saídas. Basta apenas utilizar uma matriz de pesos $\mathbf{W}$ e uma matriz de saídas $\mathbf{D}$, em que cada coluna de $\mathbf{D}$ corresponde à saída desejada de um exemplo de treinamento.

3. O Multi-Layer Perceptron

Apesar da eficiência do *Perceptron*, o trabalho de [Minsky e Papert 1969] demonstrou que esse modelo é incapaz de solucionar problemas cujas classes não sejam linearmente separáveis, ou seja, para o *Perceptron* funcionar corretamente, os exemplos a serem classificados devem estar separados suficientemente uns dos outros de maneira a garantir que a superfície de separação entre eles seja um hiperplano [Haykin 1999]. Uma solução elegante para problemas não linearmente separáveis foi o algoritmo *Back-propagation* proposto por [Rumelhart e McClelland 1986]. Ele permite o ajuste dos pesos sinápticos em redes com múltiplas camadas de neurônios totalmente conectados.

Considere a Figura 2, na qual v_j é a saída do neurônio j, dada como entrada à função de ativação associada ao neurônio, e y_j é a saída do neurônio j após a aplicação da função de ativação φ. A ideia do aprendizado do *Back-propagation* é a repetida aplicação da regra da cadeia do Cálculo, de modo a calcular a influência de cada peso sináptico da rede no processo de aprendizagem. Essa influência é calculada com relação a uma função de erro $\mathcal{E}$ de todos os parâmetros livres da rede (pesos sinápticos e bias). Assim, a função $\mathcal{E}$ é uma medida de desempenho, e o objetivo do aprendizado do *Back-propagation* é

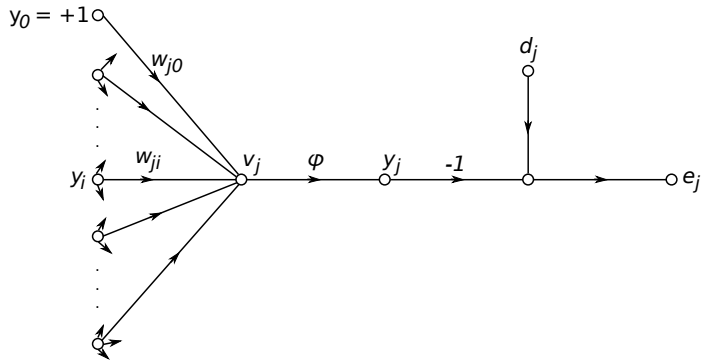

Figura 2. Fluxo do sinal em um neurônio j. Adaptado de [Haykin 1999].

ajustar esses parâmetros livres com o objetivo de minimizar a função $\mathcal{E}$. Os pesos da rede são ajustados proporcionalmente ao valor da derivada parcial $\partial\mathcal{E}/\partial w_{ji}$. Essa derivada parcial é dada na Equação 2. A derivada parcial representa um fator de sensibilidade, e determina a direção da busca pelo peso sináptico w_{ji} no espaço de pesos.

$$\frac{\partial \mathcal{E}}{\partial w_{ji}} = \frac{\partial \mathcal{E}}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \tag{2}$$

Uma vez conhecidas as derivadas parciais de cada peso sináptico, o objetivo do aprendizado é minimizar a função de erro $\mathcal{E}$ por meio da regra delta, dada pela Equação 3. Na Equação, η representa a taxa de aprendizado do algoritmo. A utilização do sinal negativo indica a execução do gradiente descendente no espaço de busca, ou seja, a procura por uma direção para a mudança do peso sináptico que reduza o erro $\mathcal{E}(t)$ [Haykin 1999].

$$w_{ji}(t+1) = w_{ji}(t) - \eta\frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)} \tag{3}$$

Basicamente, o processo de aprendizado do algoritmo *Back-propagation* consiste de dois passos: uma fase de propagação e uma fase de retropropagação. Na fase de propagação, um exemplo é aplicado aos neurônios de entrada da rede, e os sinais produzidos por esses neurônios são propagados camada a camada até que uma saída seja produzida. Após essa fase, o erro da rede é calculado subtraindo-se a saída desejada (d) da saída obtida (o). Esse erro é então retropropagado pela rede camada a camada, e os pesos são ajustados para tornar a saída da rede próxima da saída desejada. Esse ajuste é realizado da seguinte maneira: $w_{ji}^{(l)}(t+1) = w_{ji}^{(l)}(t) + \eta\delta_j^{(l)}(t)y_i^{(l-1)}(t)$.

Os gradientes locais $\delta_j^{(l)}$ denotam as derivadas da função de ativação $\varphi_j'(\cdot)$ com relação ao seu argumento. Eles são definidos da seguinte maneira:

$$\delta_j^{(l)}(t) = \begin{cases} e_j^{(L)}(t)\varphi_j'(v_j^{(L)}(t)) & \text{(1) neurônio } j \text{ na camada de saída } L \\ \varphi_j'(v_j^{(l)}(t))\sum_k \delta_k^{(l+1)}(t)w_{kj}^{(l+1)}(t) & \text{(2) neurônio } j \text{ na camada intermediária } l \end{cases}$$

4. O *Autoencoder* Simples

Os conceitos previamente apresentados podem ser utilizados para a construção de um *Autoencoder* [Hinton e Salakhutdinov 2006], uma rede neural cujo objetivo é reproduzir sua entrada na camada de saída. O modelo simples contém uma camada de entrada, uma camada intermediária, e uma camada de saída. Assim, dado um exemplo de entrada $\mathbf{x}$, o mesmo é codificado na camada intermediária e reconstruído ($\mathbf{x}'$) na camada de saída. A ideia é que a representação intermediária mantenha a informação do exemplo de entrada.

O algoritmo de treinamento é similar ao *Back-propagation*, utilizando o gradiente descendente para minimizar uma função de custo $l(f(\mathbf{x}))$. Essa função de custo vai comparar $\mathbf{x}$ e $\mathbf{x}'$ com o objetivo de avaliar o quão boa é a reconstrução. Para valores de entrada binários, pode-se utilizar $l(f(\mathbf{x})) = -\sum_k (x_k log(\hat{x}_k) + (1 - x_k) log(1 - \hat{x}_k)$. A Figura 3 ilustra um modelo simples de *Autoencoder*.

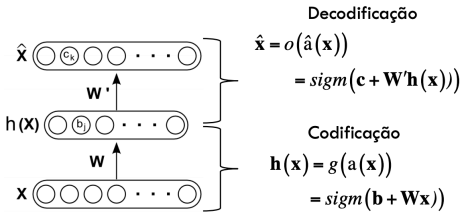

Figura 3. Ilustração do *Autoencoder* simples

5. Conclusão

Neste minicurso apresentamos brevemente três modelos de redes neurais artificiais, o *Perceptron*, o *Multi-Layer Perceptron* e o *Autoencoder*. Mais detalhes sobre as fundamentações matemáticas desses modelos estão disponibilizadas na página do minicurso[1] ministrado no ERAMIA 2020. Os respectivos algoritmos foram implementados utilizando a linguagem R [R Development Core Team 2008], e aplicados a problemas simples de classificação de dados e reconstrução de dígitos.

Referências

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition.

Hinton, G. E. e Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

McCulloch, W. e Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133.

Minsky, M. e Papert, S. (1969). *Perceptrons*. MIT Press.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rumelhart, D. E. e McClelland, J. L. (1986). *Parallel distributed processing: explorations in the microstructure of cognition*, volume 1: foundations. MIT Press.

[1] `http://www.biomal.ufscar.br/eramia2020.html`