Redes Neurais Artificiais: Backpropagation
[1]

Prof. Dr. Giancarlo D. Salton

[1] Universidade Federal da Fronteira Sul
Campus Chapecó
gian@uffs.edu.br

outubro de 2023

As redes neurais artificiais (RNAs) são modelos de *machine learning* baseados em erro. Desta forma, estes modelos são inferidos ("aprendidos") através da utilização de uma medida de erro que guia os ajustes de seus parâmetros. Desta forma, podemos formalizar algumas coisas sobre os modelos baseados em erro e, portanto, sobre as RNAs.

Dado uma *dataset* $\mathcal{D}$ com N *datapoints*: $(\mathbf{x}_n, y_n) \in (\mathcal{X}, \mathcal{Y})$, onde $\mathbf{x}_n$ são os *inputs* e y_n é o *target* relativos a um *datapoint* $n \in \mathcal{D}$, o algoritmo de aprendizado itera sobre este *dataset* e "aprende" uma função parametrizada $f : \mathcal{X} \rightarrow \mathcal{Y}$. Esta função é o modelo de *Machine learning*, onde os parâmetros ("pesos") e controlam a saída retornada pela função e, assim, descrevem a relação entre as *features* e o *target*. Para guiar o aprendizado de f, o algoritmo usa uma função de perda/custo $\mathcal{C}(y_n,\ f(\mathbf{x}_n))$, que interpretamos como o "erro" que o modelo, neste caso a RNA, comete. Esta tarefa de aprendizado do modelo também é chamado de *otimização* da função de custo, visto que estamos otimizando os parâmetros de f para que o erro seja o menor possível.

Forward Pass

Dado o *framework* acima, temos que primeiro calcular a predição que o modelo fará para um determinado exemplo que será utilizado como entrada. Utilizando o conhecimento obtido na Aula 02, podemos calcular a saída da RNA com 2 *hidden layer* da Figura 1 através das equações a seguir:

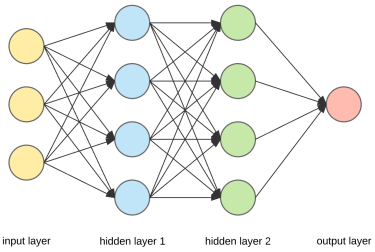

Figura 1: Representação esquemática de uma rede neural artificial com camadas de entrada (*input layer*), saída (*output layer*) e duas camadas escondidas (*hidden layer*).

$$h^{(1)} = a^1 = g\left(b^{(1)} + W^{T(1)}x\right) \tag{1}$$

$$h^{(2)} = a^2 = g\left(b^{(2)} + W^{T(2)}h^{(1)}\right) \tag{2}$$

$$o = a^3 = g\left(b^{(3)} + W^{T(3)}h^{(2)}\right) \tag{3}$$

Note que a representação das equações utiliza o formato de operação com matrizes. Além disso, a camada de entrada é representada apenas pela entrada x, uma vez que executa apenas a ativação identidade. Cada ativação a^l (também chamadas de h^l) representada a saída de uma camada l e é utilizada como entrada para a camada seguinte

$l + 1$, formando uma RNA totalmente conectada (também chamada de *fully connected*) e, portanto, utilizando camadas "densas". A função g utilizada nas equações é a função *sigmoide*[2]. Note que está é uma RNA que podes ser tanto de regressão quanto de classificação binária (*i.e.*, com duas classes). Para a continuidade deste *handout*, iremos assumir que a RNA disposta na Figura 1 é de classificação e, quando necessário, faremos um paralelo com RNAs utilizadas para regressão.

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

Backpropagation

A REDESCOBERTA DO MÉTODO DE *Backpropagation* [3] foi uma revolução na área de RNAs. Originalmente proposta na década de 1970, somente nos anos 1980 é que foi demonstrada sua utilidade no treinamento de RNAs pois possibilitou o treinamento de modelos maiores, além de possibilitar que *features* fossem aprendidas pelas RNAs. Desta forma, a RNA poderia aprender quais as partes mais relevantes dos exemplos de treinamento eram relevantes à tarefa em questão, para qual o modelo estava sendo treinado, através da análise dos erros cometidos quando fizesse uma predição.

[3] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, (323): 533–536, 1986

O algoritmo de possui um conceito simples e elegante. São apenas quatro passos para a execução completa do algoritmo. Primeiro, calcula-se o erro que a RNA comete após um *forward pass* utilizando uma função de custo[4]. Sabendo o valor do erro cometido pela RNA calcula-se o quanto cada neurônio é responsável por esse erro e faz-se um ajuste dos parâmetros de cada neurônio com base no quanto aquele neurônio é responsável pelo erro. Repete-se este processo até que o erro cometido pela rede neural seja minimizado, *i.e.*, o erro calculado para a RNA não diminua mais.

[4] Também chamada de função de perda, *cost function* ou *loss function*.

Loss Function

A FUNÇÃO DE CUSTO escolhida para calcular o erro da RNA deve ser uma função contínua e que possa ser computada a partir dos parâmetros $\mathbf{W}$ e $\mathbf{b}$ da rede neural. Desta forma, não podemos utilizar o número de acertos/erros cometidos pela RNA pois, mesmo com alterações nos parâmetros, pode ocorrer o fato de que não haja alteração na quantidade de acertos/erros que a RNA comete[5]. Assim, há uma limitação no formato das funções de custo que podem ser utilizadas para este cálculo.

[5] Pode ocorrer em alguns casos que a rede passe a acertar exemplos que o modelo errava e errar exemplos que o modelo acertava em quantidade igual. Desta forma, não há como saber se a o modelo está, de fato, melhorando suas predições ao longo do tempo.

A função de custo precisa obedecer algumas regras para que a sua aplicação seja possível no treino de uma RNA. Primeiro, a função de custo deve ser escrita como a média do custo de cada um dos *data-points* no *dataset* de treino:

$$\mathbf{C} = \frac{1}{n}\sum_{x}\mathcal{C}_x \tag{4}$$

onde: C é a função de custo; $\mathcal{C}_x$ é o cálculo do custo aplicado a um determinado exemplo x; e n é o número de exemplos d dataset de treino. A segunda regra da função de custo determina que esta deva deva ser aplicável sobre a ativação dos neurônios da camada de saída:

$$\mathbf{C} = C(a^L) = C(o) \tag{5}$$

onde: L é o índice da camada de saída; $a^L = o$ representa a ativação da camada de saída; e C é a função de custo. Perceba que para a RNA demonstrada na Figura 1 e descrita nas Equações 1, 2 e 3, $o = a^3$.

As principais funções de custo utilizadas em treino de RNAs são a *Mean Squared Error*, e a *Binary Cross entropy* e sua generalização *Binary Cross-entropy* (as vezes referenciada como *Log Loss*). A seguir iremos descrevê-las brevemente.

Mean Squared Error

A função de custo *Mean Squared Error* (MSE - as vezes chamada de *quadratic cost*) é uma função de perda utilizada também em outros algoritmos de *machine learning*, como a regressão linear, otimizadores de curva e problemas de otimização quadrática. Esta função de custo é normalmente utilizada em RNAs para tarefas de regressão[6] Sua equação[7] é definida por

$$C(y, \hat{y}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6}$$

onde: y_i é a resposta correta, contida no *dataset* de treino; $\hat{y}_i$ é a predição feita pela RNA, ou seja, é a ativação da última camada, a^L; n é o número de exemplos utilizado para o computo do MSE; e a escolha da divisão por $2n$ é um auxiliar que facilita o cálculo da derivada do MSE, como veremos adiante. Perceba também que o MSE possuí uma intuição interessante. Quando a diferença entre y_i e $\hat{y}_i$ é pequena e, portanto, o erro é pequeno (*i.e.*, quando $y_i - \hat{y}_i$ é um valor pequeno), a operação de elevar esta diferença ao quadrado resulta em um valor que normalmente não se altera muito e muitas vezes fica ainda menor[8]. Por outro lado, quando o valor do erro é um número grande, elevá-lo ao quadrado fará com que fique ainda maior[9]. Desta forma, o MSE põe mais ênfase em diferenças maiores para que o ajuste na RNA seja um ajuste maior. No caso dos erros que ficam menores, podemos

[6] Em outras palavras, o MSE é utilizado em RNAs para predição de um valor numérico dentro de um intervalo contínuo. Portanto, a camada de saída desta RNA contém apenas 1 (um) neurônio com ativação linear.

[7] O MSE também pode ser definido pela equação $\mathbf{C} = \frac{1}{2n}\sum_x \|y - a^L\|^2$. Ambas as equações são intercambiáveis.

[8] Por exemplo $(0.1)^2 = 0.01$

[9] Por exemplo, $(100)^2 = 10000$

entender como sendo um erro que está próximo do ponto correto e, portanto, deve-se proceder com cautela no ajuste deste erro.

A derivada do MSE, necessária para o processo de Gradiente Descendente é definida por:

$$\frac{\partial C}{\partial \hat{y}_i} = \frac{2}{n}(\hat{y}_i - y_i) \tag{7}$$

onde: $\frac{\partial C}{\partial \hat{y}_i}$ é a derivada com respeito à predição $\hat{y}_i$. Se utilizarmos a divisão $2n$ no cálculo do MSE, podemos simplificar esta equação da derivada para $\frac{1}{n}(\hat{y}_i - y_i)$.

Cross-Entropy

A função de custo *Cross-entropy* é amplamente utilizada em RNAs para treinamento de modelos em tarefas de classificação e também é conhecida por *log loss*[10]. O termo *cross-entropy* refere-se genericamente a uma medida de divergência entre duas distribuições de probabilidade e existem diferentes formas de cálculo, a *Binary Cross-Entropy* e sua generalização, a *Categorical Cross-Entropy*. A *Binary Cross-Entropy* calcula o custo para classificação entre duas classes distintas:

$$C(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \tag{8}$$

onde: N é o número de exemplos utilizado para o cálculo do custo; y_i é a verdadeira classe (0 ou 1) para o exemplo i; $\hat{y}_i$ é a probabilidade prevista de que o exemplo i pertence à classe 1, ou seja, $\hat{y} = a^L$.

A intuição por trás da *Binary Cross-Entropy*, é que se a saída real do neurônio estiver próxima da saída desejada para todas as entradas de treinamento x, então a entropia cruzada será próxima de zero. Para entender isso, suponha, por exemplo, que $y = 0$ e $\hat{y} \approx 0$ para alguma entrada x. Este é um caso em que o neurônio está desempenhando um bom trabalho nessa entrada. Podemos observar que o primeiro termo na Equação 8 para o custo desaparece, já que $y = 0$, enquanto o segundo termo é simplesmente $-\log(1 - a) \approx 0$. Uma análise semelhante ocorre quando $y = 1$ e $\hat{y} \approx 1$. Portanto, a contribuição para o custo será baixa desde que a saída real esteja próxima da saída desejada.

A derivada da *Binary Cross-entropy* também é uma equação simples:

$$\frac{\partial C}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{1 - y_i}{1 - \hat{y}_i} \tag{9}$$

[10] Embora estes termos sejam utilizados de forma intercambiável, a *log loss* é uma métrica específica de avaliação do desempenho de um modelo de classificação probabilística. Ela mede o erro entre as probabilidades previstas pelo modelo e as probabilidades reais associadas às classes. No entanto, como é frequentemente usada em conjunto com modelos que produzem saídas na forma de probabilidades como regressões logísticas, e as RNAs com funções de ativação sigmoid (classificação binária) ou softmax (classificação binária ou com mais classes) também geram probabilidades, confunde-se a definição da *log loss* com a *Cross-entropy*.

onde: $\frac{\partial C}{\partial \hat{y}_i}$ é a derivada parcial com respeito a predição y_i; N é o número de exemplos utilizado para o cálculo do custo; y_i é a verdadeira classe (0 ou 1) para o exemplo i; $\hat{y}_i$ é a probabilidade prevista de que o exemplo i pertence à classe 1, ou seja, $\hat{y} = a^L$.

A generalização da *Binary Cross-entropy* em *Categorical Cross-entropy* é uma medida de quão bem a distribuição de probabilidade predita pela RNA se alinha com a distribuição real dos dados quando estes estão divididos em múltiplas categorias. Esta função de custo é definida, em notação de matrizes, como[11]:

$$C(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i) \tag{10}$$

[11] Simplificando a *Categorical Cross-entropy*, podemos obter a *log loss*: $\mathbf{C} = -\frac{1}{n} \sum_{i=1}^{N} log(P(y)$.

onde: N é o número de exemplos utilizados no cálculo do custo; y_i é a predição feita para o exemplo i; $\hat{y}_{ij}$ é a probabilidade predita de que o exemplo i pertence à classe j[12]. A derivada da *Categorical Cross-entropy*, de uma RNA com ativação *softmax* na camada de saída, para uma classe j é computada como:

$$\frac{\partial C}{\partial c_j} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_{ij} - y_{ij}) \tag{11}$$

[12] A mesma equação em notação individual é definida como $-\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{ij} \cdot \log(\hat{y}_{ij})$, onde: N é o número de exemplos utilizados no cálculo do custo; M é o número de classes; y_{ij} indica se o exemplo i pertence à classe j; e $\hat{y}_{ij}$ é a probabilidade predita de que o exemplo i. pertence à classe j.

onde: $\frac{\partial C}{\partial c_j}$ é a derivada parcial com respeito a uma classe j; N é o número de exemplos utilizado para o cálculo do custo; $\hat{y}_{ij}$ é a probabilidade predita para a classe j no exemplo i; e y_{ij} indica se o exemplo i pertence à classe j (valor igual a 1) ou não (valor igual a 0).

Perceba que a escolha das funções de ativação e de custo se dão também por conveniência, pois além de serem úteis para o funcionamento das RNAs já treinadas, as suas derivadas são simples e fáceis de serem implementada.

Gradiente Descendente

Além de ser uma função contínua que seja computada a partir dos parâmetros da RNA, é desejável que a derivada da função de custo seja fácil de calcular. Isto se deve pelo fato de que o algoritmo de *Backpropagation* utiliza internamente um outro algoritmo para, a partir do erro cometido ela RNA, calcular a responsabilidade de cada neurônio e realizar os ajustes necessários em cada parâmetro. Este algoritmo, muito utilizado para otimização de funções matemáticas, é o *Gradiente Descendente*.

O algoritmo de *Gradiente Descendente* é um algoritmo muito estudado e utilizado na otimização de funções parametrizadas. Desta

forma, sua aplicação ao treinamento de RNAs se dá de forma direta, uma vez que podemos demonstrar que a ativação da camada de saída é uma função parametrizada que depende dos parâmetros relacionados aos neurônios da camada de saída e da ativação vinda da camada anterior (digamos, $l = 2$), que por sua vez depende de seus parâmetros relacionados aos seus neurônios e da ativação vinda da camada anterior ($l = 1$), e assim por diante. Perceba como é fácil demonstrar que todas as camadas estão interligadas e se conectam numa única equação se aninharmos as Equações 1, 2 e 3:

$$o = g\left(b^{(3)} + W^{T(3)}\left[g(b^{(2)} + W^{T(2)}[g(b^{(1)} + W^{T(1)}x)])\right]\right) \tag{12}$$

onde: g é a função de ativação dos neurônios; x são as *features* que descrevem os exemplos de entrada; b^i e $W^{T(i)}$ representam, respectivamente, os biases e parâmetros transpostos de uma determinada camada i; e o representa a ativação da camada de saída L da RNA (onde L é igual a $l = 3$). Por isso, o Gradiente Descendente pode ser diretamente utilizado para minimizar a função de custo relativa a RNA que está sendo treinada.

A intuição por trás do Gradiente Descente é bastante simples. Dado o custo calculado em função dos valores dos parâmetros de uma função parametrizada, deve-se procurar o ponto onde o valor do custo é o menor possível e encontraremos o valor dos parâmetros que minimizam o custo. Em outras palavras, no ponto onde o custo é mínimo encontraremos a combinação de parâmetros que, quando aplicados à função parametrizada, irá gerar aquele custo mínimo.

A derivada de uma função em um ponto específico representa a taxa de variação da função nesse ponto, indicando o quão rápido a função está aumentando ou diminuindo. Desta forma, quando calculamos as derivadas em relação à função de custo da RNA, temos a indicação de como as variações nos parâmetros do modelo aumentam ou diminuem o custo. Como o objetivo do Gradiente Descendente é encontrar o mínimo de uma função ajustando seus parâmetros na direção oposta ao gradiente, a ideia é seguir a "descida" na superfície da função para atingir o mínimo global ou local. Assim, o processo de cálculo das derivadas auxilia na compreensão de como uma função muda em relação às mudanças em seus parâmetros. Este comportamento, pode ser observado na Figura 2.

Se imaginarmos a função de custo como uma paisagem, onde a altura representa o valor da função de custo e, portanto, o erro é maior quanto mais alto estivermos nesta paisagem. O Gradiente Descendente funciona como um caminhante tentando descer para um local mais baixo e a derivada em um ponto indica a inclinação da paisagem

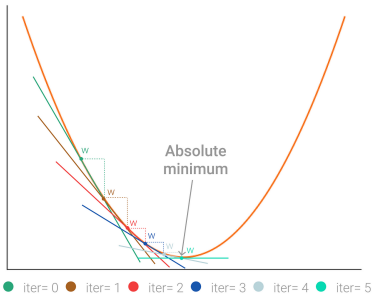

Figura 2: Representação de 4 iterações do algoritmo de Gradiente Descendente e a inclinação das derivadas em cada ponto da iteração. Perceba que, conforme nos aproximamos do fundo da parábola (gráfico da função de custo deste exemplo), as inclinação fica cada vez menor e, portanto, temos uma indicação de que estamos nos aproximando do mínimo.

naquele ponto. Se estivermos em uma posição elevada, o Gradiente Descendente nos orienta a seguir na direção em que a inclinação é mais acentuada para descer mais rapidamente. À medida que nos movemos, a derivada é recalculada em cada ponto, e a direção é ajustada com base nesse cálculo. Isso continua até que alcancemos uma região em que a derivada é próxima de zero, indicando que estamos em um mínimo local, como na Figura 3.

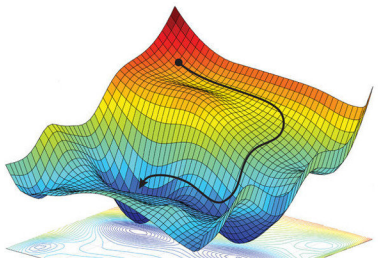

Figura 3: Exemplo de uma "paisagem" onde um caminhante precisa sair de um ponto mais algo (erro/custo alto) até um ponto mais baixo da superfície (erro/custo baixo). Um caminho hipotético para este caminhante está traçado com a linha contínua na superfície.

Atualização dos parâmetros da RNA

Seguindo a ideia do algoritmo de Gradiente Descendente, podemos descrever a regra de atualização dos parâmetros de uma função (neste caso, de uma RNA), como sendo:

$$\hat{w}_i^l = w_i^l - \alpha \times \frac{\partial \mathcal{C}}{\partial w_i^l} \tag{13}$$

onde: $\hat{w}_i^l$ é o valor atualizado de um parâmetro w_i de uma camada l; w_i^l é o valor do parâmetro w_i de uma camada l antes da atualização; α é a taxa de aprendizado[13]; e $\frac{\partial \mathcal{C}}{\partial w_i^l}$ é a derivada do erro/custo em relação ao parâmetro w_i^l.

Em notação de matrizes, a regra de atualização é escrita como:

$$\hat{\mathbf{w}}^l = \mathbf{w}^l - \alpha \odot \nabla \mathcal{C} \tag{14}$$

onde: $\hat{\mathbf{w}}^l$ é o valor atualizado dos parâmetros (vetor) da camada l; $\mathbf{w}^l$ representa os valores dos parâmetros da camada l antes da atualização; α é a taxa de aprendizado; e $\nabla \mathcal{C}$ é um vetor representando os gradientes dos parâmetros $w_{1\ldots n}^l$ da camada l, $\left(\frac{\partial \mathcal{C}}{\partial w_1}, \frac{\partial \mathcal{C}}{\partial w_2}, \cdots, \frac{\partial \mathcal{C}}{\partial w_n}, \right)^T$. Para mais detalhes, veja a Seção a seguir.

[13] A taxa de aprendizado α controla "quanto" do valor da derivada do custo é utilizado para atualizar os parâmetros da função. Se definirmos $\alpha = 1$, utilizamos 100% do valor da derivada. Se definirmos $\alpha = 0.25$, ou $\alpha = 1.10$, utilizaremos 25% ou 110%, respectivamente, do valor da derivada.

Algoritmo de Backpropagation

Sabendo que o algoritmo de Gradiente Descendente nos auxilia como um guia que aponta para onde o valor do erro é menor, utilizando derivada da função de custo, podemos aplicá-lo para treinar uma RNA utilizando o processo de *Backpropagation*.

O processo inicia com a RNA fazendo uma predição para uma entrada específica, passando esta entrada por todas as camadas, calculando as ativações de cada neurônio até obter uma saída, em um processo de *forward pass*. Em posse do erro, comparamos a saída obtida pela RNA com a saída real usando uma métrica de erro, que

neste caso é a função de custo. Esta métrica quantifica o quão longe a predição feita pela RNA está da verdade. O erro calculado é então "retropropagado" (*backpropagated*) pela rede, começando pela última camada e movendo-se em direção às camadas anteriores até atingir a camada inicial. A ideia é atribuir a cada neurônio uma parcela da responsabilidade pelo erro total cometido pelo modelo.

Com base nos erros atribuídos a cada neurônio, os pesos da rede são ajustados para reduzir o erro na próxima iteração. Este ajuste é feito usando o Gradiente Descendente, que move os pesos na direção oposta ao gradiente da função de custo[14]. Os passos de *forward pass*, cálculo do erro, retropropagação e ajuste dos pesos são repetidos iterativamente para várias entradas até que a RNA aprenda a relação entre os dados de entrada e saída. Observe este processo, demonstrado de forma simplificada na Figura 4.

Essencialmente, o *backpropagation* é como um processo de aprendizagem onde a rede se corrige iterativamente com base nos erros cometidos durante as predições. Cada neurônio "aprende" a contribuir corretamente para as predições e os pesos são ajustados para minimizar o erro cometido pela RNA. O processo se repete até que a rede seja capaz de fazer predições corretas para um conjunto diversificado de entradas.

O processo de *Backpropagation* inclui o computo de uma série de equações para realizar o treinamento da RNA. Mais especificamente: calculam-se as ativações de cada neurônio na RNA[15]; o erro cometido pela RNA, as derivadas da função de custo; a derivada da função de ativação de uma camada l em relação as suas entradas; e a derivada da função de ativação com respeito a qualquer peso (w) e com respeito a qualquer bias (b) na RNA. Este

A equação para o erro na camada de saída, comumente chamado de erro de saída ou erro de predição, é derivada da função de custo utilizada para medir o desempenho da rede neural. Essa equação é fundamental no processo de *backpropagation*, pois o erro calculado na camada de saída é usado para ajustar os pesos da rede neural durante o treinamento.

$$\delta^L = \frac{\partial C}{\partial a_i^L} \odot \sigma'(z_i^L) \tag{15}$$

onde: δ^L é o vetor de erro na camada de saída; $\frac{\partial C}{\partial a_i^L}$ é a derivada parcial da função de custo em relação à ativação da camada de saída para uma ativação i; σ' é a derivada da função de ativação da camada de saída em relação à sua transformação linear; z_i^L é a transformação linear de um neurônio da camada de saída i (*i.e.*, é a transformação linear das ativações da camada anterior). O símbolo $\odot$ representa a

[14] Como a derivada sempre aponta para o lado onde o valor da função de custo aumenta, devemos andar na direção oposta, já que é a direção onde a função de custo diminui.

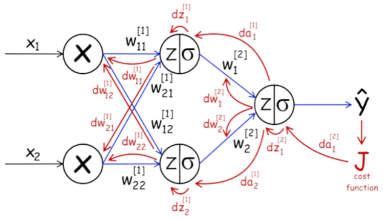

Figura 4: Exemplo simplificado do fluxo de gradientes () pela RNA durante o passo de retropropagação do algoritmo *Backpropagation*.

[15] Para mais detalhes, veja o Handout 2, "Redes Neurais Artificiais: Arquitetura".

multiplicação elemento a elemento (*elementwise*), garantindo que a multiplicação seja realizada entre os elementos correspondentes dos dois vetores.

Assim como fizemos anteriormente com as equações de ativação das camadas da RNA durante um *forward pass*, podemos expressar a equação para o erro na camada de saída em termos de notação de matrizes:

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L) \tag{16}$$

onde: $\nabla_a C$ é o gradiente da função de custo em relação à ativação da camada de saída, representado como um vetor coluna; $\odot$ denota a multiplicação elemento a elemento; $\sigma'(z^L)$ é o vetor das derivadas da função de ativação da camada de saída em relação à sua entrada ponderada. Note que mesmo representando em formato de matrizes, a derivada da função de ativação da camada de saída (σ') continua sendo uma operação elemento a elemento.

Após o cálculo do erro e sua derivada em relação à camada de saída, a próxima equação que dever ser computada é o erro em uma determinada camada ($\delta^l l$), em termos do erro na camada seguinte (δ^{l+1})[16]. Esta equação, em notação de matrizes, é expressa como:

$$\delta^l = ((W^{(l+1)})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{17}$$

onde: δ^l é o erro na camada l; δ^{l+1} é o erro na camada $l+1$; σ' é a derivada da função de ativação da camada l; e z^l é a transformação linear realizada na cama l. A derivada da função de ativação, conforme indicado por $\odot$ é uma operação elemento a elemento.

Na sequência, devemos calcular o erro em relação a qualquer peso w^l_{ij} (*i.e.*, o parâmetro conectando o neurônio i na camada $l-1$ ao neurônio j na camada l), utilizando o erro δ^l da camada l:

$$\frac{\partial C}{\partial w^l_{ij}} = a^{l-1}_i \delta^l_j \tag{18}$$

onde: $\frac{\partial C}{\partial w^l_{ij}}$ é a derivada do curso em relação ao parâmetro w^l_{ij}; a^{l-1}_i é a ativação do neurônio i da camada $l-1$; e δ^l_j é o erro do neurônio j da camada $l+1$. Essa equação representa como o erro na camada l afeta o peso específico w^l_{ij} na rede neural. Lembre-se que, durante o processo de *Backpropagation*, os pesos são ajustados proporcionalmente ao produto da ativação na camada anterior com o erro na camada atual, visando minimizar a função de custo global da rede neural. Em termos de notação de matrizes, essa equação é representado como:

[16] Lembre-se que a ativação de uma camada da RNA, depende da ativação da camada anterior: $z^l = W^{(l)} a^{(l-1)} + b^{(l)}$. Mais detalhes no Handout 2: "Redes Neurais Artificiais: Arquitetura".

$$\frac{\partial C}{\partial \mathbf{W}^l} = \mathbf{a}^{l-1}(\delta^l)^T \tag{19}$$

onde: $\frac{\partial C}{\partial \mathbf{W}^l}$ é o vetor da derivada parcial de todos os parâmetros w_{ij} da camada l; $\mathbf{a}^{l-1}$ é o vetor de ativações da camada $l-1$; e $(\delta^l)^T$ é o vetor, transposto, de erros da camada l.

Finalmente, devemos calcular o erro em relação ao todos aos bias δb^l da camada l da RNA. Este erro pode ser calculado diretamente a partir do próprio vetor de erro (δ^l):

$$\delta b^l = \delta^l \tag{20}$$

onde: δb^l é o vetor de erro em relação ao bias da camada l. Essa equação nos diz que o erro no bias é igual ao erro na ativação do neurônio correspondente na camada l. Isso é consistente com a ideia de que o bias contribui diretamente para a ativação do neurônio, e, portanto, seu erro é a mesma coisa que o erro na ativação.

Se reescrevermos a equação em notação de matrizes, este erro é representado como um vetor coluna:

$$\delta b^l = \begin{bmatrix} \delta^l_1 \\ \delta^l_2 \\ \vdots \\ \delta^l_{n^l} \end{bmatrix} \tag{21}$$

onde: δ^l_i é o erro associado ao bias i na camada l; n^l é o número de neurônios na camada l.

Considerações finais

O algoritmo de *backpropagation* é o responsável pelo treinamento de redes neurais. Resumidamente, o processo se dá através dos seguintes passos:

1. *forward pass*: A rede neural faz uma previsão para uma entrada específica, passando pela rede camada por camada até obter uma saída.

2. Cálculo do Erro: Compara-se a saída prevista com a saída real usando uma métrica de erro. Isso quantifica o quão longe a previsão da rede está da verdade.

3. *Backpropagation*: O erro calculado é então "retropropagado" (*back-propagated*) pela rede, começando pela última camada e movendo-se

em direção às camadas anteriores. A ideia é atribuir a cada neurônio uma parcela da responsabilidade pelo erro total.

4. Ajustes dos pesos: Com base nos erros atribuídos a cada neurônio, os pesos da rede são ajustados para reduzir o erro na próxima iteração. Este ajuste é feito usando o Gradiente Descendente, que move os pesos na direção oposta ao gradiente da função de custo.

5. Repetição: Os passos de *forward pass*, cálculo do erro, *backpropagation* e ajuste dos pesos são repetidos iterativamente para várias entradas até que a rede aprenda a relação entre os dados de entrada e saída.

Este algoritmo é crucial no treinamento de redes neurais e visa ajustar os pesos da RNA para minimizar a função de custo.

Referências

Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

John D. Kelleher and Brendan Tierney. *Deep Learning*. MIT Press, 1 edition, 2018.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL `http://neuralnetworksanddeeplearning.com/`.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, (323): 533–536, 1986.

Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3 edition, 2016.