Enzo Hubner Moura de Andrade
Gian Lucca Decoté Paneto Neves
Gustavo Gomes Alves
Joao Pedro Zamborlini Barcellos
Jose Augusto Costa Breciani

Backpropagation: Intuição, Derivadas e Demonstrações em Código

Serra
2025

Enzo Hubner Moura de Andrade
Gian Lucca Decoté Paneto Neves
Gustavo Gomes Alves
Joao Pedro Zamborlini Barcellos
Jose Augusto Costa Breciani

Backpropagation: Intuição, Derivadas e Demonstrações em Código

Artigo apresentado como requisito avaliativo na disciplina.

Instituto Federal do Espírito Santo

Serra
2025

Resumo

Este artigo apresenta uma abordagem teórica e prática do algoritmo de Backpropagation, destacando sua fundamentação no cálculo diferencial e na regra da cadeia. Inicialmente, discute-se a intuição matemática por trás do processo de ajuste de pesos em redes neurais artificiais. Em seguida, é apresentada uma implementação manual em Python, na qual todas as etapas do cálculo dos gradientes são explicitadas. Por fim, demonstra-se a utilização do algoritmo por meio do framework TensorFlow/Keras, evidenciando como a diferenciação automática abstrai os cálculos matemáticos. O objetivo é proporcionar uma compreensão clara da relação entre derivadas, gradientes e aprendizado de máquina.
Palavras-chave: Backpropagation. Redes neurais. Derivadas. Gradiente. Aprendizado de máquina.

Sumário

0.1 Introdução 4
0.2 Intuição do Backpropagation 4
0.3 Forward Pass 5
0.4 Backpropagation 8
0.4.1 Teoria 8
0.4.2 Exemplo de funcionamento do algoritmo 8
0.4.3 Funções utilizadas 9
0.4.4 Cálculo 9
0.4.5 Tabela de variáveis 10
0.4.6 Exemplo 10
0.4.7 Formula Final 11
0.5 Código Manual em Python 11
0.5.1 Forward Pass 11
0.5.2 Backward Pass 12
0.6 Código com PyTorch 13
0.7 Código com TensorFlow/Keras 14
0.8 Conexão com o Cálculo 14
0.9 Conclusão 15

0.1 Introdução

Redes neurais artificiais podem ser entendidas como funções matemáticas complexas, compostas por uma grande quantidade de parâmetros ajustáveis, como pesos e vieses. Esses parâmetros são os elementos internos que o modelo "aprende"a configurar durante o treinamento, determinando como a rede processa dados de entrada para gerar previsões ou decisões. Por exemplo, imagine uma rede neural como uma caixa preta que recebe informações e produz saídas. O "aprendizado"nesse contexto nada mais é do que um processo de otimização: ajustamos repetidamente esses pesos e vieses para minimizar o erro entre a saída prevista pela rede e o valor real esperado, tornando as previsões cada vez mais precisas.

Para realizar esses ajustes de forma eficiente, recorremos ao cálculo diferencial, especificamente às derivadas. As derivadas nos ajudam a medir como mudanças em um parâmetro impactam o erro total da rede. Pense nisso como calcular a "sensibilidade"de cada peça do quebra-cabeça: se eu mexer nesse peso, o erro final aumenta ou diminui? E em quanto? Essa ideia é fundamental, pois permite que a rede "aprenda"de seus erros, ajustando-se automaticamente com base em feedback matemático.

O objetivo deste trabalho é fornecer uma visão clara e acessível da ideia central do algoritmo de backpropagation, mostrando como ele utiliza derivadas e a regra da cadeia para propagar o erro de volta pela rede e otimizar os parâmetros. Além disso, será demonstrado o conceito na prática, com exemplos manuais em código e implementações em frameworks como PyTorch.

0.2 Intuição do Backpropagation

Para compreender o backpropagation de forma intuitiva, imagine uma rede neural como uma sequência de funções matemáticas aplicadas uma após a outra, formando uma cadeia de transformações. Cada neurônio na rede atua como uma pequena unidade de processamento: ele recebe valores de entrada (de dados iniciais ou de neurônios anteriores), multiplica-os por pesos, adiciona um viés (um deslocamento constante que ajusta o "limiar"de ativação do neurônio, similar a um viés inato que influencia decisões independentemente das entradas) e, por fim, aplica uma função de ativação (como a sigmoide), que introduz não-linearidade e decide se o neurônio "dispara"ou não, produzindo uma saída entre 0 e 1, por exemplo.

Agora, suponha que a rede produza uma saída errada (o erro é calculado comparando essa saída com o valor correto esperado). A questão é: como saber o quanto cada peso ou viés contribui para esse erro final? Aqui entra o backpropagation: em vez de ajustar aleatoriamente, voltamos pelos passos em ordem inversa, propagando o erro da saída de volta para as camadas anteriores.

A ferramenta matemática central para isso é a regra da cadeia. Ela permite decompor o impacto total do erro em derivadas locais simples, multiplicando-as ao longo do caminho reverso. Por exemplo, se o erro depende da saída final, que depende da ativação de uma camada intermediária, que depende de um peso inicial, a regra da cadeia calcula a derivada

0.3 Forward Pass

O *forward pass* é o cálculo direto da rede:

- a entrada é fornecida ao primeiro neurônio;
- é feita a soma ponderada e aplicada a função de ativação;
- o resultado é propagado para a próxima camada;
- ao final, obtemos uma saída.

Após a obtenção da saída, ela é comparada ao valor desejado, produzindo um erro. Esse erro será utilizado no processo de retropropagação.

O forward pass é a etapa na qual uma rede neural recebe uma entrada, realiza uma série de cálculos internos e produz uma saída. Essa etapa compreende o processo de propagação e interpretação da informação no interior da rede. Vários fatores determinam a saída da rede para uma entrada específica. São eles: Pesos; Bias; Método escolhido para o cálculo da ativação de um neurônio.

A explicação a seguir, dividida em passos, ilustra o processo através do exemplo de uma rede que busca identificar um algarismo desenhado a mão numa imagem preta e branca com tamanho 28x28. Todos os passos são executados para cada um dos neurônios de uma camada (a ser explicada), com parâmetros diferentes para cada um.

Passo 1: A imagem é convertida para um vetor x de tamanho 784x1 (784 sendo o 28^2), e cada pixel da imagem é convertido em um valor numérico e equivale a um índice em x. A cor de cada pixel determina seu valor no vetor: pixels totalmente pretos tem valor 0, pixels brancos tem valor 1, e tons intermediários equivalem a números nesse intervalo (cinza = 0.5, cinza escuro 0.1, etc.).

Passo 2: x é encaminhado para a primeira camada de neurônios da rede. Uma camada é um conjunto de números variáveis de neurônios, e cada neurônio busca um padrão específico na entrada (neste caso, na imagem). Se um neurônio encontra o padrão que busca, ele se ativa, sendo seu valor de ativação um número entre 0 e 1. Cada neurônio armazena duas informações que independem da entrada: um vetor de pesos e um bias.

Pesos (w): peso é o valor que indica o quanto uma determinada área da imagem contribui para a ativação do neurônio, e este valor é pareado com um índice no vetor x. Os pesos são armazenados num vetor w com o mesmo tamanho de x, visto que cada peso

está associado a apenas um índice (pixel) em x. O valor do peso varia com a relevância, para aquele neurônio específico, do índice ao qual ele é pareado. Se um neurônio busca, por exemplo, uma linha horizontal de 3 pixels no canto superior esquerdo da imagem, o peso atribuído ao índice desses 3 pixels em x será positivo e alto, enquanto índices de pixels em outras regiões terão pesos menores ou até negativos.

Bias (b): um único bias é atribuído a cada neurônio e determina o quanto de evidência é necessária para este sinalize que encontrou o que procurava. De forma simplificada, indica se o neurônio é exigente, permissivo ou moderado. Um bias muito negativo significa que é preciso evidência forte para sua ativação (o padrão deve ser exato ou muito próximo), enquanto um bias muito positivo significa que pouca evidência é suficiente (semelhança pode ser leve/moderada). O bias atua como um limiar ajustável: ele desloca o valor de z, tornando a ativação mais fácil ou mais difícil.

Passo 3: É calculado o valor do produto escalar (dot product) z, que determinará a ativação do neurônio, através da seguinte fórmula:

$$z = w \cdot x + b$$

Onde

x é o vetor da imagem (784 valores); w é o vetor de pesos do neurônio; b é o bias.

O produto escalar é então aplicado na função sigmoide para calcular o valor de ativação do neurônio. Se z for positivo, o resultado tende a valores maiores que 0,5; se z for negativo, tende a valores menores que 0,5.

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

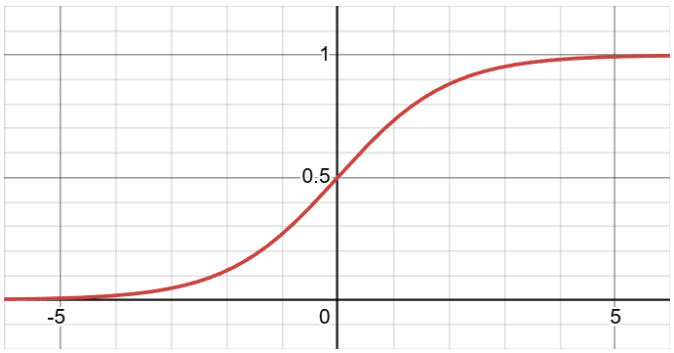


Figura 1 – Gráfico da função sigmoide

Assim, ao calcular (z), obtemos um valor positivo e mais próximo de 1 caso o produto escalar calculado pelo neurônio seja positivo e/ou alto, e mais próximo de 0 caso o produto escalar tenha sido baixo e/ou negativo, e assim obtemos o valor de ativação do neurônio.

Obs: a sigmoide não é mais utilizada em redes neurais modernas, tendo sido substituída por métodos mais eficientes. Apesar disso, foi escolhida para a explicação devido à sua facilidade de compreensão e por ter sido utilizada nos primórdios das redes neurais.

Feitos todos os passos de cálculo na primeira camada, o resultado é encaminhado para a segunda camada da seguinte forma: cada neurônio da nova camada recebe um vetor contendo as ativações de todos os neurônios da camada anterior. Assim como os neurônios da primeira camada, cada neurônio possui um conjunto de pesos, agora associados aos neurônios anteriores, e um bias. Novamente, esse neurônio calcula a soma ponderada entre os valores recebidos e seus pesos, adiciona o bias e aplica a função sigmoide, produzindo um valor de ativação entre 0 e 1.

Esse processo se repete sucessivamente, até que todas as camadas tenham sido executadas. De maneira geral:

A primeira camada detecta padrões simples (por exemplo: uma linha no meio, uma borda à esquerda, um traço na parte superior); A segunda camada recebe esses padrões simples e analisa como eles se combinam, procurando composições mais complexas; As camadas seguintes seguem a mesma lógica, examinando cada vez mais relações entre os padrões já reconhecidos.

Com isso, obtemos uma estrutura hierárquica: cada camada utiliza informações já processadas para reconhecer padrões cada vez mais específicos. Ao final desse processamento, a rede neural retorna uma saída final. No exemplo proposto, as dez saídas são agrupadas num vetor onde cada índice representa um número de 0 a 9, e cada valor corresponde à probabilidade de a imagem representar aquele número em particular. O número com maior probabilidade é o que a rede neural acha ser o correto. O resultado é então comparado com a resposta correta (label) utilizando uma função de perda ("loss function"). Esta etapa é a última do forward pass, e consiste apenas em avaliar o desempenho da rede sem alterar seus parâmetros, e permite que seja iniciado o processo de retropropagação (backpropagation) para que os parâmetros sejam ajustados e a rede "aprenda".

Vamos considerar que o número a ser identificado pela rede era 5, logo, o resultado produziu deveria ser [0, 0, 0, 0, 0, 1, 0, 0, 0, 0], e o resultado produzido foi [0.02, 0.01, 0.88, 0.04, 0.01, 0.01, 0.02, 0.003, 0.005, 0.005] (os índices são contados à partir de 0). O índice 2 possui a maior probabilidade calculada, o que indica que o valor que a rede neural "acha" ser o certo é o 2. Porém, sabemos que o número correto era, na verdade, 5, logo a rede está errada ao produzir este resultado. Para calcular o valor da perda (loss), ou seja, o quão longe a rede está do resultado correto, utilizamos a função Binary Cross-Entropy (BCE), definida como

$$L = -[y \log(p) + (1 - y) \log(1 - p)]$$

onde

L é a perda; y é o valor esperado para determinada saída (no caso, y de 5 deveria ser 1 e dos demais 0); p é o valor produzido pela rede.

Quando o valor para determinada saída deve ser 1, temos $L = -\log(p)$, ou seja, quanto maior a probabilidade predita, menor a perda; Quando o valor deve ser 0, $L = -\log(1-p)$, ou seja, quanto menor a probabilidade, menor a perda.

Após calcular a BCE individualmente para cada saída da rede, os valores são somados e divididos pelo número de saídas (média), produzindo um único valor de perda que representa o erro global daquela predição. Esse valor será usado pelo backpropagation para ajustar pesos e biases.

0.4 Backpropagation

0.4.1 Teoria

Como a função de perda recebe como input a saída de uma rede neural, e essa saída é uma função composta formada por muitas funções de ativação aninhadas de neurônios individuais, a diferenciação da função de perda envolve a diferenciação de toda a rede. Para isso, a backpropagation usa a regra da cadeia.

Começando pela camada final, o backpropagation diferencia a função de perda para calcular como cada parâmetro individual da rede contribui para o erro geral de uma única entrada.

Cada neuronio terá um valor de 0 a 1. Em um modelo perfeitamente treinado, o neuronio com a classificação correta estaria muito próximo de 1.

0.4.2 Exemplo de funcionamento do algoritmo

Vamos chamar o neuronio de saída de Lc. A função de ativação de Lc é uma função composta, contendo muitas funções aninhadas de toda a rede neural. Minimizar a função de perda implica em ajustar toda a rede para que a saída da função se aproxime cada vez mais de 1.

Para se aproximar, é necessário encontrar as derivadas parciais da função de ativação de Lc. Pois elas mostram como alterações nas camadas anteriores impactam na saída.

A saída da função de ativação de Lc depende das contribuições da camada imediatamente antes dela, no caso, L - 1. Uma das formas de alterar a saída é modificar o peso entre esses neuronios. Ao calcular a derivada parcial de cada peso de L - 1, em relação a outros pesos, podemos ver como aumentar ou diminuir algum deles trará um melhor resultado.

É importante lembrar que não é só o neuronio anterior que impacta no resultado. Na verdade, o erro pode ter acontecido bem antes. É nesse caso que o backpropagation entra de fato em ação. Com ele é possível diferenciar as funções de ativação de L - 1 em relação

a L - 2, depois em L - 3 e assim por diante, repetindo recursivamente até chegar na camada de entrada. Quando esse processo é terminado, teremos o gradiente da função de perda que é um vetor de sua derivada parcial para cada parâmetro de peso e vies na rede.

Com o gradiente em mãos é possível minimizar a função de perda, e assim otimizar o modelo. Mover o gradiente para baixo diminuirá a perda, pois, a partir das derivadas parciais, sabemos em qual direção mover os parametros para reduzir a perda.

0.4.3 Funções utilizadas

$$z = i(w_i x_i) + b$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$E = \frac{1}{2}(t - y)^2$$

z: Soma de todos as entradas ponderadas pelos pesos, mais o vies;

y: Função de ativação, que vai gerar a saida do neuronio;

E: Erro do Quadrado médio. t é o valor a ser alcançado;

0.4.4 Cálculo

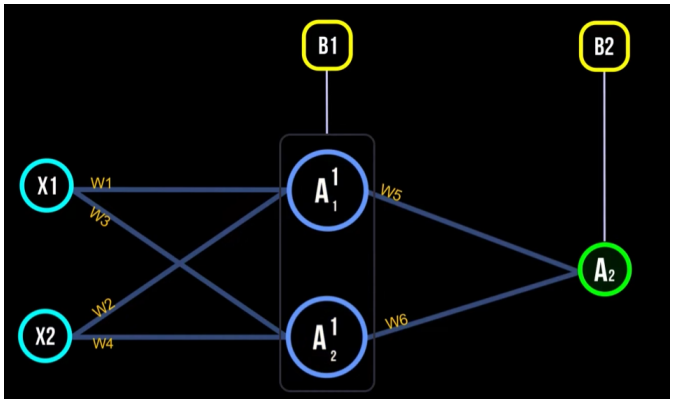


Figura 2 – Gráfo em análise

O neuronio A11 produz a soma ponderada $(w1 \cdot X1 + w2 \cdot X2 + b1)$ e uma função de ativação (sigmoide) é aplicada a essa soma $= (w1 \cdot X1 + w2 \cdot X2 + b1)$.

O neuronio A21 produz a soma ponderada $(w3 \cdot X1 + w4 \cdot X2 + b1)$ e uma função de ativação (sigmoide) é aplicada a essa soma $= (w3 \cdot X1 + w4 \cdot X2 + b1)$.

Os valores de saida de cada neuronio serão armazenados em um vetor A1.

No neuronio A2 produz a soma ponderada $(w5 \cdot a11 + w6 \cdot a21 + b2)$ e uma função de ativação (sigmoide) é aplicada a essa soma $= (w5 \cdot a11 + w6 \cdot a21 + b2)$. Essa informação é armazenada em um vetor A2.

Expandindo:

$$(w5 \cdot (w1 \cdot X1 + w2 \cdot X2 + b1) + w6 \cdot (w3 \cdot X1 + w4 \cdot X2 + b1) + b2)$$

Como tem uma função dentro da outra, e para encontrar os valores é necessário derivar, então usamos a regra da cadeia.

0.4.5 Tabela de variáveis

Xi: Entradas

Zj1: Entrada Ponderada Camada Oculta $Z11 = (w1 \cdot X1 + w2 \cdot X2 + B1)$

Aj1: Saída Camada Oculta $\sigma(Zj1)$

Z2: Entrada Ponderada Camada de Saída $(w5 \cdot A11 + w6 \cdot A21 + B2)$

A2: Saída Final da Rede $\sigma(Z2)$

0.4.6 Exemplo

Encontrar a responsabilidade do peso w1 no Erro, ou seja, encontrar $\frac{dE}{dw1}$. Para isso é preciso traçar o caminho que os conecta.

$$E \to A2 \to Z2 \to A11 \to Z11 \to w1$$

Gradiente do peso: pela regra da cadeia, a derivada é a multiplicação de todas as derivadas parciais ao longo do caminho.

$$\frac{dE}{dw1} = \frac{dE}{dA2} \cdot \frac{dA2}{dZ2} \cdot \frac{dZ2}{dA11} \cdot \frac{dA11}{dZ11} \cdot \frac{dZ11}{dw1}$$

Desenvolvendo:

Sensibilidade do erro a saída (Erro do Quadrado Médio):

$$\frac{dE}{dA2} = -(t - A2)$$

Sensibilidade da Saída em relação a entrada ponderada:

$$\frac{dA2}{dZ2} = A2(1 - A2)$$

Propagação do erro pelo peso:

Como $Z2 = (w5 \cdot A11 + w6 \cdot A21 + B2)$, a derivada em relação a A11 é:

$$\frac{dZ2}{dA11} = w5$$

Multiplicando os termos 1, 2 e 3, obtemos a sensibilidade do Erro total a saída do neurônio A11:

$$\frac{dE}{dA11} = \frac{dE}{dA2} \cdot \frac{dA2}{dZ2} \cdot \frac{dZ2}{dA11}$$

Sensibilidade da saida oculta a sua entrada ponderada:

$$\frac{dA11}{dZ11} = A11(1 - A11)$$

Sensibilidade da Entrada Ponderada ao Peso w1:

Como $Z11 = (w1 \cdot X1 + w2 \cdot X2 + B1)$:

$$\frac{dZ11}{dw1} = X1$$

0.4.7 Formula Final

$$\frac{dE}{dw1} = [-(t - A2) \cdot A2(1 - A2) \cdot w5 \cdot A11(1 - A11) \cdot X1]$$

Isso representa o quanto e em qual direção o peso w1 deve ser ajustado para reduzir o Erro E.

0.5 Código Manual em Python

Nesta seção é apresentada uma implementação manual do algoritmo. Para compreender plenamente esse código, é importante relacionar cada parte com os conceitos teóricos de propagação direta (forward pass) e retropropagação do erro (backward pass), descritos anteriormente.

0.5.1 Forward Pass

O *forward pass* é a etapa onde os dados percorrem a rede neural no sentido das entradas para as saídas. Matematicamente, cada camada realiza duas operações fundamentais:

1. Cálculo da entrada ponderada:

$$z = W \cdot x + b$$

onde W são os pesos, x é o vetor de entrada e b é o viés.

2. Aplicação da função de ativação:

$$a = \sigma(z)$$

No caso, a sigmoide, responsável por introduzir não-linearidade.

Essas duas operações aparecem exatamente no código abaixo:

```
def forward(self, entradas):
    self.z_oculta = np.dot(entradas, self.pesos) + self.vieses
    self.ativacao_oculta = self.sigmoid(self.z_oculta)

    self.z_saida = np.dot(self.ativacao_oculta, self.pesos_saida) + self.vies_saida
    self.ativacao_saida = self.sigmoid(self.z_saida)

    return self.ativacao_saida
```

Esse processo corresponde matematicamente a:

$$A_1 = \sigma(W_1 X + b_1)$$

$$A_2 = \sigma(W_2 A_1 + b_2)$$

O resultado final é a saída da rede para a entrada fornecida.

0.5.2 Backward Pass

O *backward pass* implementa a retropropagação do erro. Seu objetivo é calcular os gradientes necessários para ajustar pesos e vieses, aplicando a regra da cadeia no caminho reverso da rede:

$$\frac{dE}{dW} = \frac{dE}{dA} \cdot \frac{dA}{dZ} \cdot \frac{dZ}{dW}$$

No código manual, isso é traduzido diretamente para operações matriciais:

```
def backward(self, entradas, saidas_esperadas):
    erro_saida = saidas_esperadas - self.ativacao_saida
    gradiente_saida = erro_saida * self.derivada_sigmoid(self.ativacao_saida)

    gradiente_pesos_saida = np.dot(self.ativacao_oculta.T, gradiente_saida)
    gradiente_vies_saida = np.sum(gradiente_saida, axis=0, keepdims=True)

    erro_oculto = np.dot(gradiente_saida, self.pesos_saida.T)
    gradiente_oculto = erro_oculto * self.derivada_sigmoid(self.ativacao_oculta)

    gradiente_pesos_ocultos = np.dot(entradas.T, gradiente_oculto)
    gradiente_vieses_ocultos = np.sum(gradiente_oculto, axis=0, keepdims=True)
```

Cada linha tem significado matemático:

- **erro_saida**: diferença $t - A_2$
- **gradiente_saida**: aplica derivada da sigmoide
- **erro_oculto**: propagação reversa via pesos W_2
- **gradiente_oculto**: derivada local da ativação na primeira camada

Isso implementa exatamente:

$$\delta_2 = -(t - A_2) \cdot A_2(1 - A_2)$$
$$\delta_1 = (\delta_2 W_2^T) \cdot A_1(1 - A_1)$$

E os gradientes finais tornam-se:

$$\frac{\partial E}{\partial W_2} = A_1^T \delta_2$$
$$\frac{\partial E}{\partial W_1} = X^T \delta_1$$

0.6 Código com PyTorch

O PyTorch encapsula automaticamente o cálculo do gradiente utilizando um mecanismo chamado **Autograd**. Esse sistema constrói dinamicamente um *computational graph* contendo todas as operações realizadas no forward pass. Cada operação sabe calcular sua própria derivada.

Assim, quando chamamos:

```
loss.backward()
```

O PyTorch percorre esse grafo no sentido reverso e aplica automaticamente a regra da cadeia, calculando:

$$\nabla_W E,\ \nabla_b E,\ \nabla_X E$$

para todos os tensores que possuem `requires_grad=True`.

A grande diferença para o código manual é:

- Não precisamos implementar derivadas.
- Não precisamos multiplicar manualmente os termos da regra da cadeia.
- A biblioteca acumula gradientes para cada parâmetro.

Assim, o PyTorch automatiza todo o processo que foi explicitamente codificado na versão manual.

0.7 Código com TensorFlow/Keras

No TensorFlow/Keras, o processo é semelhante ao PyTorch, porém realizado dentro de um *GradientTape*. Durante o forward pass, o TensorFlow grava todas as operações. Depois, ao calcular o erro, ele retropropaga automaticamente os gradientes.

A rede usada no seu exemplo:

```
self.modelo = keras.Sequential([
    layers.Dense(3, activation='sigmoid', input_shape=(2,)),
    layers.Dense(1, activation='sigmoid')
])
```

corresponde matematicamente a:

$$A_1 = \sigma(W_1 X + b_1) \qquad A_2 = \sigma(W_2 A_1 + b_2)$$

Durante o treinamento:

1. O TensorFlow computa o forward pass.
2. Avalia a função de perda.
3. Traça o grafo computacional automaticamente.
4. Usa o *GradientTape* para aplicar a regra da cadeia.
5. Atualiza pesos usando o otimizador escolhido (SGD, Adam, RMSProp etc.).

Ou seja, o TensorFlow abstrai completamente as operações manuais de derivação feitas no início da seção.

0.8 Conexão com o Cálculo

O procedimento realizado corresponde à aplicação da derivação em funções compostas, conforme estabelecido no cálculo diferencial.

Uma rede neural pode ser interpretada como uma função parametrizada, composta por múltiplas camadas e dependente de diversas variáveis internas — especificamente, os pesos e vieses que definem seu comportamento.

O algoritmo de backpropagation consiste essencialmente no cálculo sistemático das derivadas parciais dessa função composta, utilizando a regra da cadeia para determinar a sensibilidade do erro em relação a cada parâmetro.

O conjunto dessas derivadas parciais constitui o gradiente, que fornece a direção de maior variação do erro e, consequentemente, orienta o ajuste dos parâmetros durante o processo de otimização.

A função de perda, portanto, recebe como entrada não apenas a saída da rede, mas, indiretamente, todas as transformações feitas ao longo das camadas.

Minimizar a perda significa ajustar os pesos para reduzir esse erro, e isso exige saber como cada peso individual afeta o valor final da função de perda. Em termos matemáticos, precisamos calcular derivadas parciais da perda em relação a cada peso:

$$\frac{\partial E}{\partial w_1}$$

O problema é que nenhum peso afeta o erro de forma direta: ele influencia o erro apenas por meio de várias funções intermediárias. Assim, para encontrar essas derivadas, é necessário aplicar a regra da cadeia repetidamente ao longo da rede.

O backpropagation é justamente esse processo sistemático de aplicação da regra da cadeia para percorrer a rede neural no sentido inverso — da saída de volta até a entrada. Ele começa na camada final determinando a sensibilidade do erro em relação à saída do último neurônio; em seguida, utiliza essa informação para calcular a sensibilidade em relação às ativações da camada anterior, e assim sucessivamente, camada por camada. A cada passo, calcula-se como pequenas mudanças nas ativações e nos pesos anteriores influenciam o resultado final.

Em outras palavras, o algoritmo determina, para cada caminho possível dentro da rede,

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial a_L} \cdot \frac{\partial a_L}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdot \ldots \cdot \frac{\partial z_1}{\partial w}$$

multiplicando todas as derivadas locais entre cada par de funções conectadas. Esse encadeamento mostra como uma pequena variação em um peso específico modifica, direta ou indiretamente, a perda observada na saída.

Ao final do processo, o que obtemos é o gradiente da função de perda: um vetor contendo a derivada parcial da perda em relação a cada peso e a cada viés da rede. Esse gradiente fornece a direção exata em que a função de perda cresce mais rapidamente. Para otimizar a rede, movemos os parâmetros na direção contrária — a direção de descida do gradiente — diminuindo assim o erro.

Desse modo, a conexão entre cálculo e redes neurais fica clara: backpropagation é essencialmente um exercício de derivação de funções compostas usando a regra da cadeia, aplicado de forma estruturada e eficiente. Ele traduz o comportamento da rede em termos matemáticos precisos, permitindo que cada peso seja ajustado de forma proporcional à sua responsabilidade no erro final. Essa combinação entre composição funcional e cálculo diferencial é o que torna o aprendizado de redes neurais possível.

0.9 Conclusão

Backpropagation é essencialmente cálculo diferencial aplicado. Ele permite ajustar pesos utilizando derivadas de forma sistemática. Implementações manuais ajudam a

entender o mecanismo, enquanto frameworks automatizam os cálculos.