BACK PROPOGATION ALGORITHM

GRADIENT OF LOSS FUNCTION:

The loss function is a method of evaluating how well your machine learning algorithm models your featured data set.
In other words, loss functions are a measurement of how good your model is in terms of predicting the expected outcome.
Loss function refer to the training process that uses back-propagation to minimize the error between the actual and predicted outcome.

Here we mainly derive the gradient expressions of the mean square error loss function and the cross-entropy loss function.The mean square error loss function expression is:

$$L=\frac{1}{2}\sum_{k=1}^{K}(y_k-o_k)^2$$

Then its partial derivative is:

$$\frac{\partial L}{\partial o_i}=\frac{1}{2}\sum_{k=1}^{K}\frac{\partial}{\partial o_i}(y_k-o_k)^2$$

Decomposition by the law of derivative of composite function:

$$\frac{\partial L}{\partial o_i}=\frac{1}{2}\sum_{k=1}^{K}2\cdot(y_k-o_k)\cdot\frac{\partial(y_k-o_k)}{\partial o_i}$$

That is:

$$\frac{\partial L}{\partial o_i}=\sum_{k=1}^{K}(y_k-o_k)\cdot-1\cdot\frac{\partial o_k}{\partial o_i}$$

$$=\sum_{k=1}^{K}(o_k-y_k)\cdot\frac{\partial o_k}{\partial o_i}$$

Considering that $\frac{\partial o_k}{\partial o_i}$ is 1 when $k=i$ and $\frac{\partial o_k}{\partial o_i}$ is 0 for other cases, that is, the partial derivative $\frac{\partial L}{\partial o_i}$ is only related to the *i*th node, so the summation symbol in the preceding formula can be removed. The derivative of the mean square error function can be expressed as:

$$\frac{\partial L}{\partial o_i}=(o_i-y_i)$$

GRADIENT OF CROSS ENTROPY FUNCTION:

When calculating the cross-entropy loss function, the Softmax function and the cross-entropy function are generally implemented in a unified manner.
We first derive the gradient of the Softmax function, and then derive the gradient of the cross-entropy function.
Gradient of Softmax: The expression of Softmax:

$$p_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

We know that if

$$f(x) = \frac{g(x)}{h(x)}$$

The derivative of the function is:

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{h(x)^2}$$

For Softmax function, $g(x) = e^{z_i}$, $h(x) = \sum_{k=1}^{K} e^{z_k}$. We'll derive its gradient at two conditions: $i = j$ and $i \neq j$.

- $i = j$. The derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{e^{z_i} \sum_{k=1}^{K} e^{z_k} - e^{z_j} e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{e^{z_i}\left(\sum_{k=1}^{K} e^{z_k} - e^{z_j}\right)}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{\left(\sum_{k=1}^{K} e^{z_k} - e^{z_j}\right)}{\sum_{k=1}^{K} e^{z_k}}$$

The preceding expression is the multiplication of p_i and $1 - p_j$, and $p_i = p_j$. So when $i = j$, the derivative of Softmax $\frac{\partial p_i}{\partial z_j}$ is:

$$\frac{\partial p_i}{\partial z_j} = p_i\left(1 - p_j\right), i = j$$

- $i \neq j$. Extend the Softmax function:

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}}{\partial z_j} = \frac{0 - e^{z_j} e^{z_i}}{\left(\sum_{k=1}^{K} e^{z_k}\right)^2}$$

$$= \frac{-e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \times \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

That is:

$$\frac{\partial p_i}{\partial z_j} = -p_j \cdot p_i$$

Gradient of cross-entropy function Consider the expression of the cross-entropy loss function:

$$L = -\sum_k y_k \, log \, log \left(p_k\right)$$

Here we directly derive the partial derivative of the final loss value L to the logits variable z_i of the network output, which expands to:

$$\frac{\partial L}{\partial z_i} = -\sum_k y_k \frac{\partial \, log \, log \left(p_k\right)}{\partial z_i}$$

Decompose the composite function *log log h* into:

$$= -\sum_k y_k \frac{\partial \, log \, log \left(p_k\right)}{\partial p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

That is:

$$= -\sum_k y_k \frac{1}{p_k} \cdot \frac{\partial p_k}{\partial z_i}$$

where $\frac{\partial p_k}{\partial z_i}$ is the partial derivative of the Softmax function that we have derived.

Split the summation symbol into the two cases: $k = i$ and $k \neq i$, and substitute the expression of $\frac{\partial p_k}{\partial z_i}$, we can get:

$$\frac{\partial L}{\partial z_i} = -y_i(1 - p_i) - \sum_{k \neq i} y_k \frac{1}{p_k}(-p_k \cdot p_i)$$

$$= -y_i(1 - p_i) + \sum_{k \neq i} y_k \cdot p_i$$

$$= -y_i + y_i p_i + \sum_{k \neq i} y_k \cdot p_i$$

That is:

$$\frac{\partial L}{\partial z_i} = p_i \left(y_i + \sum_{k \neq i} y_k \right) - y_i$$

In particular, the one-hot encoding method for the label in the classification problem has the following relationship:

$$\sum_k y_k = 1$$

$$y_i + \sum_{k \neq i} y_k = 1$$

Therefore, the partial derivative of cross-entropy can be further simplified to:

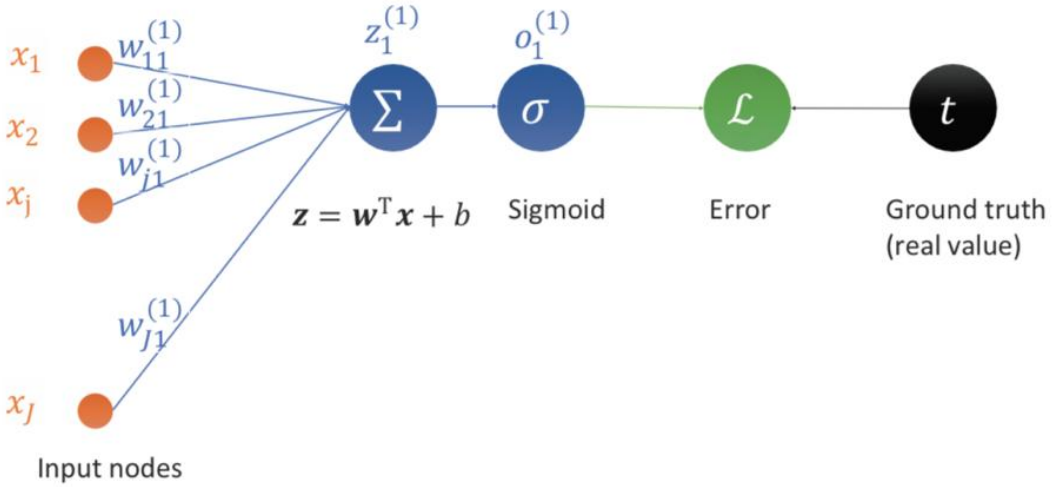
$$\frac{\partial L}{\partial z_i} = p_i - y_i$$

GRADIENT OF FULLY CONNECTED NETWORK:

GRADIENT OF SINGLE NEURON:

- For a neuron model using Sigmoid activation function, its mathematical model can be written as:

  O(1)=sigma(w(1)^T+b(1))
- The superscript of the variable represents the number of layers. For example, o(1) represents the output of the first layer and x is the input of the network
- We take the partial derivative derivation ∂L/ ∂wj1 of the weight parameter wj1 as an example
- The number of input nodes is J. The weight connection from the input of the jth node to the output o(1) is denoted as wj1 (1) , where the superscript indicates the number of layers to which the weight parameter belongs, and the subscript indicates the starting node number and the ending node number of the current connection.
- For example, the subscript j1 indicates the jth node of the previous layer to the first node of the current layer.
- The variable before the activation function σ is called z1(1) , and the variable after the activation function σ is called o1(1) . Because there is only one output node, so o1(1)=o(1)=o .
- The error value L is calculated by the error function between the output and the real label



- The loss can be expressed as:

  L =1/2(o1(1) - t)^2 = ½(o(1) - t)^2
- Among them, t is the real label value. Adding 1/2 does not affect the direction of the gradient, and the calculation is simpler
- We take the weight variable wj1 of the jth (j ∈ [1,J]) node as an example and consider the partial derivative of the loss function L:

$$\partial L/\partial wj1 = (o1 - t)\,\partial o1/\partial wj1$$

- Considering o1 = σ(z1) and the derivative of the Sigmoid function is σ′ = σ(1 − σ), we have:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)\frac{\partial \sigma(z_1)}{\partial w_{j1}}$$

$$= (o_1 - t)\sigma(z_1)(1 - \sigma(z_1))\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

Write $\sigma(z_1)$ as o_1:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)\frac{\partial z_1^{(1)}}{\partial w_{j1}}$$

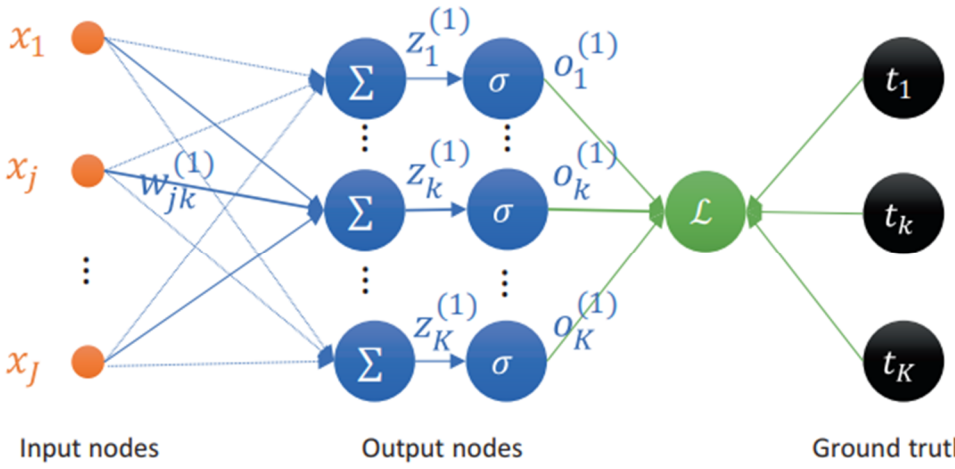
Consider $\frac{\partial z_1^{(1)}}{\partial w_{j1}} = x_j$, we have:

$$\frac{\partial L}{\partial w_{j1}} = (o_1 - t)o_1(1 - o_1)x_j$$

- It can be seen from the preceding formula that the partial derivative of the error to the weight wj1 is only related to the output value o1, the true value t, and the input xj connected to the current weight

GRADIENT OF FULLY CONNECTED LAYER

- We generalize the single neuron model to a single-layer network of fully connected layers, as shown in Figure .
- The input layer obtains the output vector o(1) through a fully connected layer and calculates the mean square error with the real label vector t.
- The number of input nodes is J, and the number of output nodes is K.



- The multi-output fully connected network layer model differs from the single neuron model in that it has many more output nodes o1(1) ,o2(1) ,o3(1) ,ok(1) …… , and each output node corresponds to a real label t1,

t2, …, tK. wjk is the connection weight of the jth input node and the kth output node. The mean square error can be expressed as:

- $$L = \frac{1}{2}\sum_{i=1}^{K}\left(o_i^{(1)} - t_i\right)^2$$

- Since ∂L / ∂wjk is only associated with node ok (1) , the summation symbol in the preceding formula can be removed, that is, i = k:

- $$\frac{\partial L}{\partial w_{jk}} = \left(o_k - t_k\right)\frac{\partial o_k}{\partial w_{jk}}$$

- Substitute ok = σ(zk):

- $$\frac{\partial L}{\partial w_{jk}} = \left(o_k - t_k\right)\frac{\partial \sigma\left(z_k\right)}{\partial w_{jk}}$$

Consider the derivative of the Sigmoid function $\sigma' = \sigma(1-\sigma)$:

$$\frac{\partial L}{\partial w_{jk}} = \left(o_k - t_k\right)\sigma\left(z_k\right)\left(1-\sigma\left(z_k\right)\right)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Write $\sigma(z_k)$ as o_k:

$$\frac{\partial L}{\partial w_{jk}} = \left(o_k - t_k\right)o_k\left(1-o_k\right)\frac{\partial z_k^{(1)}}{\partial w_{jk}}$$

Consider $\frac{\partial z_k^{(1)}}{\partial w_{jk}} = x_j$:

- $$\frac{\partial L}{\partial w_{jk}} = \left(o_k - t_k\right)o_k\left(1-o_k\right)x_j$$

- It can be seen that the partial derivative of wjk is only related to the output node ok (1) of the current connection, the label tk (1) of the corresponding true, and the corresponding input node xj.

$$\text{Let } \delta_k = (o_k - t_k)o_k(1 - o_k),\ \frac{\partial L}{\partial w_{jk}} \text{ becomes:}$$

$$\frac{\partial L}{\partial w_{jk}} = \delta_k x_j$$

- The variable δk characterizes a certain characteristic of the error gradient propagation of the end node of the connection line.

- After using the representation δk, the partial derivative ∂L/ ∂wjk is only related to the start node xj and the end node δk of the current connection.

- Now that the gradient propagation method of the single-layer neural network (i.e., the output layer) has been derived, next we try to derive the gradient propagation method of the penultimate layer.

- After completing the propagation derivation of the penultimate layer, similarly, the gradient propagation mode of all hidden layers can be derived cyclically to obtain gradient calculation expressions of all layer parameters.

CHAIN RULE:

The chain rule is a fundamental concept in calculus that allows you to find the derivative of a composite function. In other words, it helps you calculate the rate of change of a function that is composed of two or more functions nested inside each other. The chain rule is especially important in calculus when dealing with functions where one quantity depends on another, which in turn depends on another, and so on.

Mathematically, the chain rule is stated as follows:

If you have a composite function y = f(g(x)), where:

y is the final output or dependent variable,

f(u) is the outer function,

g(x) is the inner function, and u = g(x),

Then, the derivative of y with respect to x, denoted as dy/dx, is calculated as:

dy/dx = (df/du) * (du/dx)

In words, the chain rule says that to find the derivative of the composite function y = f(g(x)), you multiply the derivative of the outer function f(u) with respect to its variable u by the derivative of the inner function g(x) with respect to x.

Here's a more concrete example:

Let y = f(u) = u^2, and u = g(x) = 3x - 1. We want to find dy/dx.

Find df/du: df/du = 2u.

Find du/dx: du/dx = 3.

Now, apply the chain rule:

dy/dx = (df/du) * (du/dx)

dy/dx = (2u) * (3)

Since u = g(x), we can substitute u back in:

dy/dx = (2(3x - 1)) * 3

dy/dx = 6(3x - 1)

So, dy/dx = 18x - 6.

The chain rule is a powerful tool that enables you to find derivatives of complex functions by breaking them down into simpler functions and considering how changes in the inner function affect the outer function. It's a fundamental concept in calculus and is widely used in various fields of mathematics and science.

CHAIN RULE IN GRADIENT PROPOGATION:

In the context of machine learning and neural networks, the chain rule plays a crucial role in gradient propagation during the training process. Gradient propagation is the process of computing gradients (derivatives) of a composite function, such as a neural network, with respect to its parameters. The chain rule is used to calculate these gradients efficiently, allowing the model to update its parameters during training through optimization algorithms like gradient descent.

Let's break down how the chain rule is used in gradient propagation:

Neural Network Forward Pass:

During the forward pass of a neural network, input data is processed through multiple layers. Each layer applies an activation function to its input and produces an output. This process can be represented as a sequence of functions:

Input data: x

Layer 1: $h_1 = f_1(W_1 * x + b_1)$

Layer 2: $h_2 = f_2(W_2 * h_1 + b_2)$

...

Output layer: $y = f_o(W_o * h_k + b_o)$

Here, each layer has its own weights (W) and biases (b), and f_1, f_2, ..., f_o are activation functions.

Computing Loss:

The neural network makes predictions (y) based on the input data, and the predictions are compared to the actual target values to compute a loss function (e.g., Mean Squared Error or Cross-Entropy Loss).

Gradient Calculation - Backpropagation:

The goal of training is to minimize the loss. To do this, you need to calculate the gradients of the loss with respect to the parameters (weights and biases) in each layer of the network. This is where the chain rule comes into play.

Starting from the output layer and moving backward through the network (hence the term "backpropagation"), you calculate the gradients layer by layer using the chain rule. The key steps are as follows:

Compute the gradient of the loss with respect to the output layer's inputs.

Propagate this gradient backward through each layer, multiplying it by the gradient of the layer's inputs with respect to its parameters, using the chain rule.

Mathematically, the chain rule is applied as follows:

$\partial Loss/\partial W_o = \partial Loss/\partial y * \partial y/\partial(W_o * h_k + b_o)$

$\partial Loss/\partial b_o = \partial Loss/\partial y * \partial y/\partial(W_o * h_k + b_o)$

$\partial Loss/\partial W_2 = \partial Loss/\partial y * \partial y/\partial(W_2 * h_1 + b_2) * \partial h_1/\partial(W_2 * h_1 + b_2) * \partial(W_2 * h_1 + b_2)/\partial W_2$

$\partial Loss/\partial b_2 = \partial Loss/\partial y * \partial y/\partial(W_2 * h_1 + b_2) * \partial h_1/\partial(W_2 * h_1 + b_2) * \partial(W_2 * h_1 + b_2)/\partial b_2$

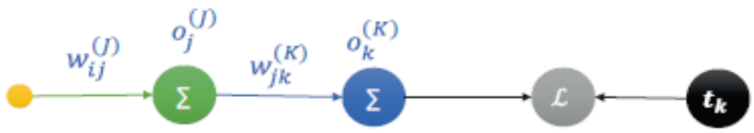
...

The chain rule is repeatedly applied for each layer to calculate the gradients.

Updating Parameters:

Once you have computed the gradients of the loss with respect to the parameters, you can use them to update the parameters through an optimization algorithm like gradient descent. The goal is to adjust the parameters to minimize the loss, thereby improving the model's performance.

In summary, the chain rule is a fundamental concept in gradient propagation within neural networks. It enables the efficient calculation of gradients for each layer, allowing the network to learn and adapt its parameters during training. This process is critical for the successful training of machine learning models, including deep neural networks.



). Gradient propagation illustration

BACK PROPOGATION ALGORITHM:

Backpropagation, short for "backward propagation of errors," is an algorithm used to train artificial neural networks, including deep learning models. It is a key component of the training process and is responsible for updating the model's weights to minimize the error between predicted and actual values. Here is a step-by-step explanation of the backpropagation algorithm:

Step 1: Initialize Weights and Biases

Initialize the weights and biases of the neural network. These values are typically initialized randomly.

Step 2: Forward Pass

Input data is fed forward through the network layer by layer, from the input layer to the output layer.

For each layer:

Calculate the weighted sum of inputs for each neuron in the layer.

Apply the activation function to the weighted sum to get the output of each neuron.

Step 3: Compute Loss

Calculate the loss (error) between the predicted output and the actual target values using a suitable loss function (e.g., Mean Squared Error for regression or Cross-Entropy Loss for classification).

Step 4: Backward Pass (Backpropagation)

Compute the gradient of the loss with respect to the output layer's inputs. This gradient measures how much a small change in the output of the network affects the loss.

∂Loss/∂output_layer_inputs

Propagate this gradient backward through the network to calculate the gradients of the loss with respect to the weights and biases of each layer. This is done using the chain rule.

∂Loss/∂weights and ∂Loss/∂biases for each layer

Update the weights and biases of each layer using an optimization algorithm like gradient descent. The goal is to adjust these parameters in a way that minimizes the loss.

Step 5: Repeat

Repeat steps 2-4 for a fixed number of iterations (epochs) or until the loss converges to a satisfactory level.

Step 6: Evaluate Model

After training, evaluate the model's performance on a separate validation dataset or test dataset to assess its generalization ability.

Step 7: Use the Model

Once the model is trained and evaluated, it can be used for making predictions on new, unseen data.

This was simple process… lets understand it in detail

Backpropagation is the core of how neural networks learn. Up until this point, you learned

that training a neural network typically happens by the repetition of the following three steps:

• Feedforward: get the linear combination (weighted sum), and apply the activation

function to get the output prediction ($\hat{y}$):

$\hat{y} = \sigma \cdot W(3) \cdot \sigma \cdot W(2) \cdot \sigma \cdot W(1) \cdot (x)$

• Compare the prediction with the label to calculate the error or loss function:

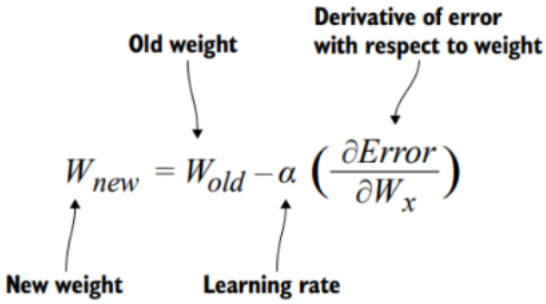
$E(W, b) = |\hat{y}i - yi|$

• Use a gradient descent optimization algorithm to compute the Δw that optimizes the

error function:

$\Delta wi = -\alpha \, dE/dw$

• Backpropagate the Δw through the network to update the weights:



Backpropagation, or backward pass, means propagating derivatives of the error with respect to each specific weight dE/dWi .

from the last layer (output) back to the first layer (inputs) to adjust weights. By propagating the change in weights Δw backward from the prediction node (ŷ) all the way through the hidden layers and back to the input layer, the weights get updated:

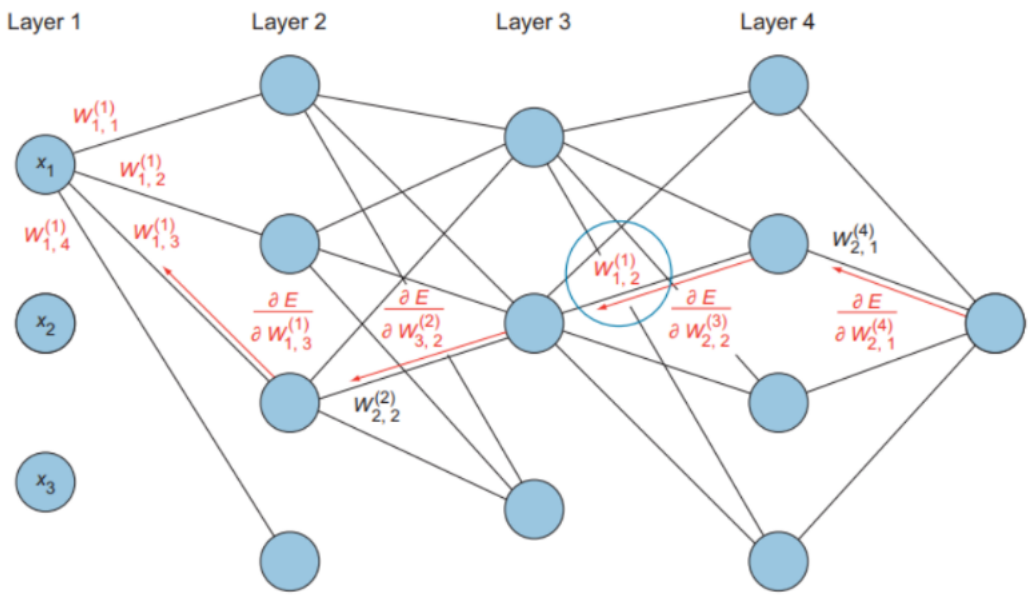
(wnext – step = wcurrent + Δw)

This will take the error one step down the error mountain. Then the cycle starts again (steps 1 to 3) to update the weights and take the error another step down, until we get to the minimum error.

Backpropagation might sound clearer when we have only one weight. We simply adjust the weight by adding the Δw to the old weight

wnew = w – α dE/dwi.

But it gets complicated when we have a multilayer perceptron (MLP) network with many weight variables. To make this clearer, consider the scenario in figure:



How do we compute the change of the total error with respect to dE/dw13 ?

How much will the total error change when we change the parameter w13?

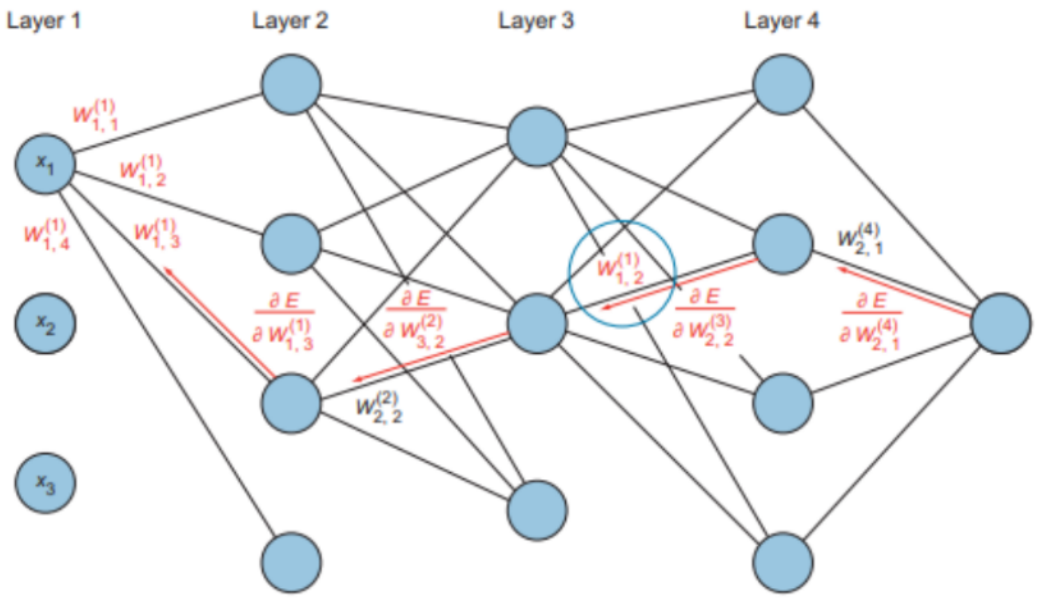
how to compute by applying the derivative rules on the error function.

That is straightforward because w21 is directly connected to the error function. But to compute the derivatives of the total error with respect to the weights all the way back to the input, we need a calculus rule called the chain rule.

Let’s apply the chain rule to calculate the derivative

of the error with respect to the third weight on the first input w1,3 (1) , where the (1) means layer 1, and w1,3 means node number 1 and weight number 3:

$$\frac{dE}{dw_{1,3}^{(1)}} = \frac{dE}{dw_{2,1}^{(4)}} \times \frac{dw_{2,1}^{(4)}}{dw_{2,2}^{(3)}} \times \frac{dw_{2,2}^{(3)}}{dw_{3,2}^{(2)}} \times \frac{dw_{3,2}^{(2)}}{dw_{1,3}^{(1)}}$$



The error back propagated to the

edge w1,3 (1) = effect of error on edge 4 · effect on edge 3 ·effect on edge 2 · effect on target edge .

Thus the backpropagation technique is used by neural networks to update the weights to solve

the best fit problem.

HANDS ON:

This example will demonstrate a single-layer neural network for a simple regression problem using backpropagation.

Let's start with a simple neural network with one neuron and one input feature. We'll implement forward and backward passes for training.

```
import numpy as np


# Define the sigmoid activation function and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))
```

```
def sigmoid_derivative(x):
    return x * (1 - x)

# Define the neural network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases with random values
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, x):
        # Forward pass
        self.hidden_input = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.output = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        return self.output

    def backward(self, x, y, output):
        # Backpropagation
        self.output_error = y - output
        self.output_delta = self.output_error
        self.hidden_error = self.output_delta.dot(self.weights_hidden_output.T)
        self.hidden_delta = self.hidden_error * sigmoid_derivative(self.hidden_output)
        self.weights_hidden_output += self.hidden_output.T.dot(self.output_delta)
        self.weights_input_hidden += x.T.dot(self.hidden_delta)
```

```
        self.bias_output += np.sum(self.output_delta, axis=0)
        self.bias_hidden += np.sum(self.hidden_delta, axis=0)

    def train(self, x, y, epochs):
        for _ in range(epochs):
            output = self.forward(x)
            self.backward(x, y, output)

if __name__ == "__main__":
    # Define the dataset
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])

    # Create and train the neural network
    input_size = 2
    hidden_size = 4
    output_size = 1
    nn = NeuralNetwork(input_size, hidden_size, output_size)

    epochs = 10000
    nn.train(X, y, epochs)

    # Test the trained network
    for i in range(len(X)):
        prediction = nn.forward(X[i])
        print("Input:", X[i], "Actual:", y[i], "Predicted:", prediction)
```

This code defines a simple neural network with one hidden layer and uses the sigmoid activation function. It trains the network using the XOR dataset.

EXAMPLE 2:

Here's an example of a four-layer fully connected neural network implemented in Python for binary classification using backpropagation. This network has two input nodes, three hidden layers with 20, 50, and 25 nodes respectively, and two output nodes for binary classification:

```
import numpy as np


# Define the sigmoid activation function and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):

    return x * (1 - x)


# Define the neural network class

class NeuralNetwork:

    def __init__(self, input_size, hidden_sizes, output_size):

        self.input_size = input_size

        self.hidden_sizes = hidden_sizes

        self.output_size = output_size

        self.num_layers = len(hidden_sizes) + 2  # Including input and output layers


        # Initialize weights and biases with random values

        self.weights = [np.random.rand(input_size, hidden_sizes[0])]

        self.biases = [np.zeros((1, hidden_sizes[0]))]


        for i in range(len(hidden_sizes) - 1):
```

```
            self.weights.append(np.random.rand(hidden_sizes[i], hidden_sizes[i+1]))
            self.biases.append(np.zeros((1, hidden_sizes[i+1])))

        self.weights.append(np.random.rand(hidden_sizes[-1], output_size))
        self.biases.append(np.zeros((1, output_size)))

    def forward(self, x):
        self.layer_outputs = []
        input_layer = x

        # Forward pass through hidden layers
        for i in range(self.num_layers - 1):
            weighted_sum = np.dot(input_layer, self.weights[i]) + self.biases[i]
            layer_output = sigmoid(weighted_sum)
            self.layer_outputs.append(layer_output)
            input_layer = layer_output

        return input_layer

    def backward(self, x, y, output):
        # Backpropagation
        deltas = [None] * self.num_layers
        error = y - output
        delta = error * sigmoid_derivative(output)
        deltas[-1] = delta

        # Calculate deltas for hidden layers
```

```
        for i in range(self.num_layers - 2, 0, -1):
            error = deltas[i + 1].dot(self.weights[i].T)
            delta = error * sigmoid_derivative(self.layer_outputs[i - 1])
            deltas[i] = delta

        # Update weights and biases
        for i in range(self.num_layers - 1):
            self.weights[i] += self.layer_outputs[i - 1].T.dot(deltas[i])
            self.biases[i] += np.sum(deltas[i], axis=0)

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            for i in range(len(X)):
                x = X[i]
                target = y[i]
                output = self.forward(x)
                self.backward(x, target, output)

    def predict(self, x):
        return self.forward(x)

if __name__ == "__main__":
    # Define the dataset
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0, 1], [1, 0], [1, 0], [0, 1]])  # One-hot encoded labels

    # Create and train the neural network
```

```
input_size = 2
hidden_sizes = [20, 50, 25]
output_size = 2
learning_rate = 0.1
epochs = 10000

nn = NeuralNetwork(input_size, hidden_sizes, output_size)
nn.train(X, y, epochs, learning_rate)

# Test the trained network
for i in range(len(X)):
    prediction = nn.predict(X[i])
    print("Input:", X[i], "Actual:", y[i], "Predicted:", prediction)
```