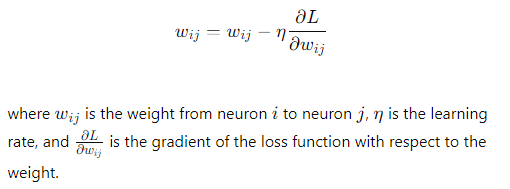
**Backpropagation**, short for "backward propagation of errors," is an algorithm used to train neural networks. It is a supervised learning technique that adjusts the weights of the network to minimize the error between the predicted output and the actual output.

#### Key Concepts in Backpropagation

1. **Neural Network Structure**:
   * **Input Layer**: The initial layer where the network receives input data.
   * **Hidden Layers**: Intermediate layers where computations are performed.
   * **Output Layer**: The final layer that produces the prediction.
2. **Forward Propagation**:
   * Data flows from the input layer through the hidden layers to the output layer.
   * Each neuron calculates a weighted sum of its inputs and applies an activation function to produce an output.
   * The output of each neuron becomes the input for the next layer.
3. **Error Calculation**:
   * The error is calculated as the difference between the predicted output and the actual output.
   * A loss function, such as Mean Squared Error (MSE) or Cross-Entropy Loss, is used to quantify the error.
4. **Backward Propagation**:
   * The error is propagated back through the network to update the weights.
   * The algorithm calculates the gradient of the loss function with respect to each weight by applying the chain rule of calculus.
   * This process involves two main steps:
     + **Gradient Calculation**: Compute the gradient of the loss with respect to each weight.
     + **Weight Update**: Adjust the weights in the opposite direction of the gradient to reduce the error. The weight update rule is typically:



#### Steps in the Backpropagation Algorithm

1. **Initialize Weights and Biases**:
   * Start with small random values for weights and biases.
2. **Forward Pass**:
   * Compute the output of the network for a given input by passing the data through each layer.
3. **Compute Error**:
   * Calculate the error using the loss function.
4. **Backward Pass (Backpropagation)**:
   * Compute the gradient of the loss with respect to each weight.
   * Propagate the error back through the network, layer by layer, starting from the output layer.
5. **Update Weights and Biases**:
   * Adjust the weights and biases using the gradients and the learning rate.
6. **Repeat**:
   * Perform steps 2-5 for a specified number of iterations or until the error converges to a small value.

The provided code implements a simple backpropagation algorithm for training a neural network using NumPy. Here's an analysis and explanation of each part:

**Code Breakdown**

1. **Data Preparation:**

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([89], [86], [92]), dtype=float)

X = X / np.amax(X, axis=0)

y = y / 100

* + X is the input data, a 3×23 \times 23×2 matrix.
  + y is the output data, a 3×13 \times 13×1 matrix.
  + Both X and y are normalized.

1. **Activation Functions:**

def sigmoid(x):

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

def derivatives\_sigmoid(x):

return x \* (1 - x)

* + sigmoid(x): The sigmoid activation function.
  + derivatives\_sigmoid(x): The derivative of the sigmoid function, used for backpropagation.

1. **Hyperparameters and Network Structure:**

epoch = 7000

lr = 0.1

inputlayer\_neurons = 2

hiddenlayer\_neurons = 3

output\_neurons = 1

* + epoch: Number of training iterations.
  + lr: Learning rate.
  + Network structure: 2 input neurons, 3 hidden neurons, 1 output neuron.

1. **Weight and Bias Initialization:**

wh = np.random.uniform(size=(inputlayer\_neurons, hiddenlayer\_neurons))

bh = np.random.uniform(size=(1, hiddenlayer\_neurons))

wout = np.random.uniform(size=(hiddenlayer\_neurons, output\_neurons))

bout = np.random.uniform(size=(1, output\_neurons))

* + Random initialization of weights (wh, wout) and biases (bh, bout).

1. **Training Process (Backpropagation):**

for i in range(epoch):

hinp1 = np.dot(X, wh)

hinp = hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1 = np.dot(hlayer\_act, wout)

outinp = outinp1 + bout

output = sigmoid(outinp)

E0 = y - output

outgrad = derivatives\_sigmoid(output)

d\_output = E0 \* outgrad

EH = d\_output.dot(wout.T)

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

wout += hlayer\_act.T.dot(d\_output) \* lr

* + Forward pass:
    - Compute hidden layer input hinp1 and apply bias hinp.
    - Apply sigmoid activation to get hidden layer output hlayer\_act.
    - Compute output layer input outinp1 and apply bias outinp.
    - Apply sigmoid activation to get final output output.
  + Backward pass:
    - Calculate error E0 between actual output y and predicted output output.
    - Compute gradient for output layer d\_output.
    - Calculate error propagated to the hidden layer EH.
    - Compute gradient for hidden layer d\_hiddenlayer.
    - Update weights wout using the gradients.

1. **Printing Results:**

print("Input:\n" + str(X))

print("actual output:\n" + str(y))

print("predicted output:\n", output)

* + Print the normalized input X, actual output y, and predicted output output.

**Explanation**

This script demonstrates the fundamental steps of training a neural network with one hidden layer using backpropagation. It includes forward propagation to compute the outputs and backpropagation to update the weights based on the errors. The network is trained over a specified number of epochs with a given learning rate.

**Key Points**

1. **Normalization:** The inputs and outputs are normalized to ensure that the network trains effectively.
2. **Sigmoid Activation:** The sigmoid function is used for the hidden and output layers.
3. **Backpropagation:** Errors are propagated backward through the network to update the weights.
4. **Gradient Calculation:** Gradients are calculated using the derivative of the sigmoid function.
5. **Weight Update:** Weights are updated using the calculated gradients and learning rate.