|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

Experiment No. 05

|  |  |
| --- | --- |
| Semester | B.E. Semester VIII – Computer Engineering |
| Subject | Deep Learning Lab |
| Subject Professor In-charge | Prof. Kavita Shirsat |
| Academic Year | 2024-25 |

|  |  |
| --- | --- |
| Student Name | Deep Salunkhe |
| Roll Number | 21102A0014 |

**Title:** Backpropagation algorithm to train a fully connected DNN

**Explanation:**

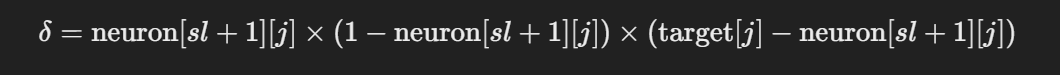
The bp function implements the **backpropagation algorithm**, which is a key component of training artificial neural networks. It is used to adjust the weights of the network based on the error between the predicted output and the actual target values. The function follows these steps:

**Step 1: Initialize Required Data Structures**

* nm: A new map to store updated weights.
* deltas: A map to store error gradients (partial derivatives) for weight updates.
* nta: The learning rate (set to 1 in this case).
* nl: Total number of layers in the neural network.
* sl: Index of the second-last layer (output layer is at nl-1, so second-last is nl-2).

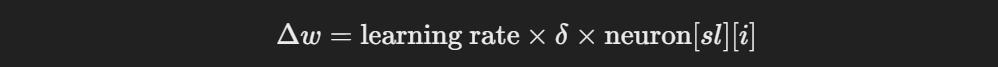
**Step 2: Compute Output Layer Error**

For the **output layer**, the gradient (delta) is computed as:



This formula comes from differentiating the loss function (usually Mean Squared Error or Cross-Entropy) with respect to the neuron activation.

* **For each neuron j in the output layer (sl+1)**:
  + Compute the delta value.
  + Store it in the deltas map.
  + Update the corresponding weight using:

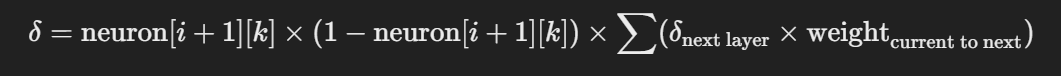


* The updated weight is stored in nm.

**Step 3: Compute Hidden Layer Errors**

For hidden layers (i going from sl-1 to 0):

* The delta for each neuron is computed using the **chain rule** by propagating the error backward.



* The **sum term** accumulates contributions from all neurons in the next layer (i+2).
* The weight update follows the same formula as in the output layer.
* Store the updated weights in nm.

**Step 4: Update Weights**

Finally, replace the original weight matrix m with nm, ensuring the updated weights are used in the next forward pass.

**Implementation:**

**#include** <iostream>

**#include** <vector>

**#include** <map>

**#include** <math.h>

**using** **namespace** std;

**void** printMap(map<pair<pair<**int**, **int**>, pair<**int**, **int**>>, **float**> **&**m)

{

**for** (**const** **auto** **&**entry : m)

    {

**const** **auto** **&**key **=** entry.first;

**const** **auto** **&**value **=** entry.second;

        // Printing the key: pair<pair<int, int>, pair<int, int>>

        cout **<<** "Key: ((" **<<** key.first.first **<<** ", " **<<** key.first.second **<<** "), ("

**<<** key.second.first **<<** ", " **<<** key.second.second **<<** ")) ";

        // Printing the value

        cout **<<** "Value: " **<<** value **<<** endl;

    }

}

**void** takeweights(map<pair<pair<**int**, **int**>, pair<**int**, **int**>>, **float**> **&**m, vector<**int**> neuronCount)

{

**int** nl **=** neuronCount.size();

**for** (**int** i **=** 0; i **<** nl **-** 1; i**++**)

    {

**for** (**int** j **=** 0; j **<** neuronCount**[**i**]**; j**++**)

        {

**for** (**int** k **=** 0; k **<** neuronCount**[**i **+** 1**]**; k**++**)

            {

                cout **<<** "Enter weight of " **<<** i **<<** "-" **<<** j **<<** "to" **<<** i **+** 1 **<<** "-" **<<** k **<<** endl;

**float** t;

                cin **>>** t;

                m**[**{{i, j}, {i **+** 1, k}}**]** **=** t;

            }

        }

    }

}

**void** inputlayer(vector<vector<**float**>> **&**neuron, vector<**int**> neuronCount)

{

**int** inls **=** neuronCount**[**0**]**;

    cout **<<** "Enter input layer" **<<** endl;

**for** (**int** i **=** 0; i **<** inls; i**++**)

    {

**float** t;

        cin **>>** t;

        neuron**[**0**]**.push\_back(t);

    }

}

// void ff(vector<vector<float>>neuron,vector<int>neuronCount,  map<pair<pair<int,int>,pair<int,int>>,float>m,vector<float>bias){

//  int nn=neuronCount.size();

//  for(int i=0;i<nn-1;i++){

//      for(int j=0;j<neuronCount[i+1];j++){

//          float cal=0;

//          for(int k=0;k<neuronCount[i];k++){

//               cal+=neuron[i][k]\*m[{{i,k},{i+1,j}}];

//          }

//          cal=cal+bias[i+1];

//          neuron[i+1].push\_back(1/(1+pow(2.71828,-cal)));

//          cout<<neuron[i+1].back()<<endl;

//      }

//      cout<<"hi"<<endl;

//      cout<<neuron[i+1].size()<<endl;

//  }

// }

**void** ff(vector<vector<**float**>> **&**neuron, vector<**int**> **&**neuronCount, map<pair<pair<**int**, **int**>, pair<**int**, **int**>>, **float**> **&**m, vector<**float**> **&**bias)

{

**int** nn **=** neuronCount.size();

    // Make sure all layers except the first are pre-sized (if you want to add elements dynamically)

**for** (**int** i **=** 1; i **<** nn; i**++**)

    {

        neuron**[**i**]**.clear(); // Clear the previous layer values, if any

    }

**for** (**int** i **=** 0; i **<** nn **-** 1; i**++**)

    {

**for** (**int** j **=** 0; j **<** neuronCount**[**i **+** 1**]**; j**++**)

        {

**float** cal **=** 0;

            // Calculate weighted sum for the current neuron in the next layer

**for** (**int** k **=** 0; k **<** neuronCount**[**i**]**; k**++**)

            {

                cal **+=** neuron**[**i**][**k**]** **\*** m**[**{{i, k}, {i **+** 1, j}}**]**; // weight from layer i to layer i+1

            }

            // Add bias for the current neuron in the next layer

            cal **=** cal **+** bias**[**i **+** 1**]**;

            // Apply the sigmoid activation function

**float** output **=** 1 **/** (1 **+** exp(**-**cal)); // Sigmoid function: 1 / (1 + e^(-x))

            // Push the output of the neuron in the next layer

            neuron**[**i **+** 1**]**.push\_back(output);

            cout **<<** neuron**[**i **+** 1**]**.back() **<<** endl; // Print the output of the current neuron

        }

        // cout << "hi" << endl;

        // cout << neuron[i + 1].size() << endl;  // Print the size of the next layer

    }

}

**void** printNeuron(vector<vector<**float**>> neuron, vector<**int**> neuronCount)

{

**int** n **=** neuronCount.size();

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

**int** ns **=** neuron**[**i**]**.size();

**for** (**int** j **=** 0; j **<** ns; j**++**)

        {

            cout **<<** neuron**[**i**][**j**]** **<<** " ";

        }

        cout **<<** endl;

    }

}

// void printNeuron(const vector<vector<float>>& neuron, const vector<int>& neuronCount) {

//     int n = neuronCount.size();

//

//     // Check if the size of neuron matches neuronCount

//     for(int i = 0; i < n; i++) {

//         int ns = neuronCount[i];

//

//         // Ensure that the number of neurons in each layer is consistent with the neuronCount

//         if (neuron[i].size() != ns) {

//             cout << "Error: Mismatch in neuron size for layer " << i << endl;

//             return;  // Exit the function if there's an inconsistency

//         }

//

//         for(int j = 0; j < ns; j++) {

//             cout << neuron[i][j] << " ";

//         }

//         cout << endl;

//     }

// }

**void** bp(vector<vector<**float**>> neuron, map<pair<pair<**int**, **int**>, pair<**int**, **int**>>, **float**> **&**m, vector<**int**> neuronCount, vector<**float**> target)

{

    map**<**pair**<**pair**<int**, **int>**, pair**<int**, **int>>**, **float>** nm;

    map**<**pair**<**pair**<int**, **int>**, pair**<int**, **int>>**, **float>** deltas;

**int** nta **=** 1;

    // for output layer edges

**int** nl **=** neuronCount.size(); // number of layer

**int** sl **=** nl **-** 2;

**for** (**int** i **=** 0; i **<** neuronCount**[**sl**]**; i**++**)

    {

**for** (**int** j **=** 0; j **<** neuronCount**[**sl **+** 1**]**; j**++**)

        {

**float** deltat **=** neuron**[**sl **+** 1**][**j**]** **\*** (1 **-** neuron**[**sl **+** 1**][**j**]**) **\*** (target**[**j**]** **-** neuron**[**sl **+** 1**][**j**]**);

            deltas**[**{{sl, i}, {sl **+** 1, j}}**]** **=** deltat;

**float** deltaw **=** nta **\*** deltat **\*** neuron**[**sl**][**i**]**;

            nm**[**{{sl, i}, {sl **+** 1, j}}**]** **=** m**[**{{sl, i}, {sl **+** 1, j}}**]** **+** deltaw;

        }

    }

    // for remaning edges

**for** (**int** i **=** sl **-** 1; i **>=** 0; i**--**)

    {

**for** (**int** j **=** 0; j **<** neuronCount**[**i**]**; j**++**)

        {

**for** (**int** k **=** 0; k **<** neuronCount**[**i **+** 1**]**; k**++**)

            {

**float** thatsum **=** 0;

**for** (**int** t **=** 0; t **<** neuronCount**[**i **+** 2**]**; t**++**)

                {

                    thatsum **+=** deltas**[**{{i **+** 1, k}, {i **+** 2, t}}**]** **\*** m**[**{{i **+** 1, k}, {i **+** 2, t}}**]**;

                }

**float** deltat **=** neuron**[**i **+** 1**][**k**]** **\*** (1 **-** neuron**[**i **+** 1**][**k**]**) **\*** thatsum;

                deltas**[**{{i, j}, {i **+** 1, k}}**]** **=** deltat;

**float** deltaw **=** nta **\*** deltat **\*** neuron**[**i**][**j**]**;

                nm**[**{{i, j}, {i **+** 1, k}}**]** **=** m**[**{{i, j}, {i **+** 1, k}}**]** **+** deltat;

            }

        }

    }

    printMap(nm);

    m **=** nm;

}

**int** main()

{

**int** hl;

    cout **<<** "Enter number of layer" **<<** endl;

    cin **>>** hl;

    vector**<int>** neuronCount(hl);

    cout **<<** "Enter number of neuron in each layer" **<<** endl;

**for** (**int** i **=** 0; i **<** hl; i**++**)

    {

        cin **>>** neuronCount**[**i**]**;

    }

    vector**<**vector**<float>>** neuron(hl);

    cout **<<** "Enter bias" **<<** endl;

    vector**<float>** bias(hl);

**for** (**int** i **=** 0; i **<** hl; i**++**)

    {

**float** t;

        cin **>>** t;

        bias**[**i**]** **=** t;

    }

    vector**<float>** target(neuronCount**[**hl **-** 1**]**);

    cout **<<** "Enter targer" **<<** endl;

**for** (**int** i **=** 0; i **<** neuronCount**[**hl **-** 1**]**; i**++**)

    {

        cin **>>** target**[**i**]**;

    }

    map**<**pair**<**pair**<int**, **int>**, pair**<int**, **int>>**, **float>** m;

    takeweights(m, neuronCount);

    printMap(m);

    inputlayer(neuron, neuronCount);

    cout **<<** "first forward pass" **<<** endl;

    ff(neuron, neuronCount, m, bias);

    printNeuron(neuron, neuronCount);

    cout **<<** "first backward pass" **<<** endl;

    bp(neuron, m, neuronCount, target);

    cout **<<** "second forward pass" **<<** endl;

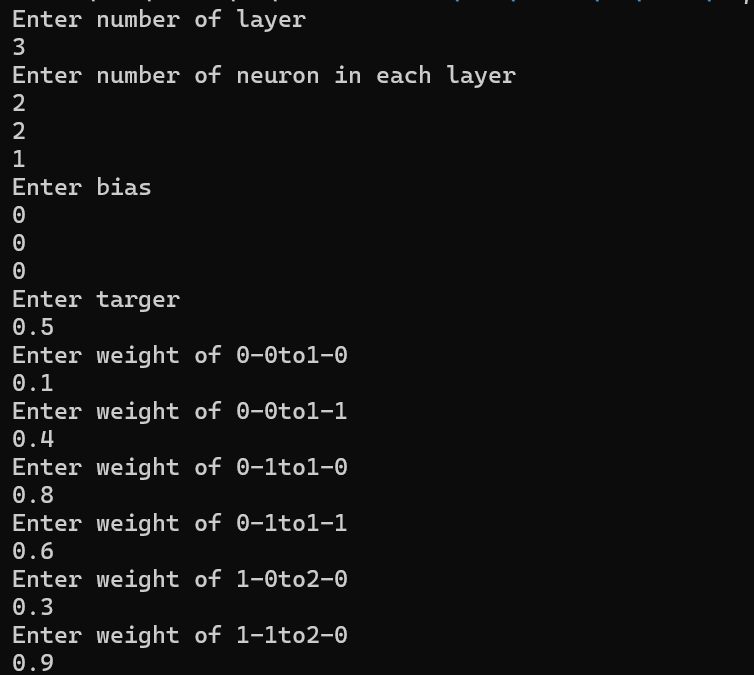
    ff(neuron, neuronCount, m, bias);

    printNeuron(neuron, neuronCount);

**return** 0;

}

**Output:**

****

