

# Neurális hálózatok

András Mamenyák<sup>1</sup> and Roland Bamli<sup>1</sup>

<sup>1</sup>Mérnök informatikus (BSc) szakos hallgató, Debreceni Egyetem

2013. december 5.

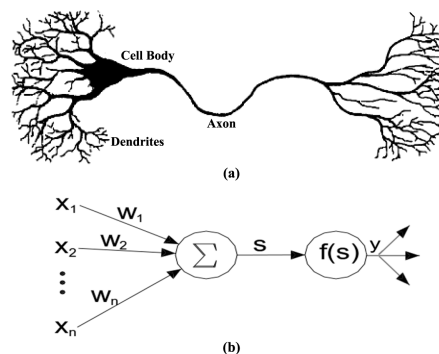
## 1. Bevezetés

### 1.1. A neurális hálózatok kialakulása

A neurális hálózatok a mesterséges intelligencia egy típusa, amelyet az állatok központi idegrendszere, különösen az agy ihletett, amely képes a tanulásra, a mintafelismerésre is. Megalkotásához biológiai ismeretekre és az idegsejt működésének pontosabb megismerésére volt szükség. Ez csak a 20. században valósult meg. Az első neuron modellt 1947-ben alkotta meg McCulloch és Pitts, az első mesterséges neuront pedig Rosenblatt 1958-ban. A neurális hálózatok egy ígéretes, új tudományterület, mely Webos 1974-es "back propagation" algoritmusával és annak 1986-os újra felfedezése után indult igazán fejlődésnek.

### 1.2. A mesterséges neuron felépítése, működése

Egy mesterséges neuron, mint a biológiai, több bemenettel és egy kimenettel rendelkezik (1. ábra). Egy általános neuron működése szerint meghatározza a bemenetek súlyozott összegét és ezen végrehajt valamilyen nem lineáris leképezést. Ez utóbbit nevezik aktivációs, transzfer vagy aktiváló függvénynek. A végeredmény pedig a neuron kimeneti jele. Egy másik változat a lineáris összegzést megvalósító neuron, amikor nem történik lineáris leképezés.



1. ábra. A biológiai neuron (a) és a mesterséges neuron (b) összehasonlítása

A 1. ábrán a neuron bemeneteit  $x_i$  jelöli, a kimeneti jel pedig  $y$ . Először a bemenetek súlyozott összegei kerülnek meghatározásra:

$$s = \sum_{i=0}^n W_i \cdot x_i = W^T \cdot x$$

Abban az esetben, ha a neuron lineáris összegzést valósít meg, ezzel már meg is kaptuk a kimeneti jelet:

$$y = s = W^T \cdot x$$

Nem lineáris esetben szükség van még a nem lineáris leképezésre. Ebben az esetben a neuron kimeneti jele a következő:

$$y = f(s) = f(W^T \cdot x)$$

ahol  $f(s)$  az aktivizációs függvény. Erre a célra a négy leggyakrabban használt függvény a lépcső- vagy szignumfüggvény, a „telítésszerű lineáris” függvény, a tangens hiperbolicus függvény és a szigmoid függvény.

Használunk egy másik elterjedt neuron típust is a RBF (Radial Bass Function) hálózatokban. Ennél a típusnál nincs lineáris összegzés, az összes bemenet az aktivizációs függvénybe kerül, mely több bemenet esetén több változós függvény lesz.

### 1.3. A neuron hálózatok felépítése

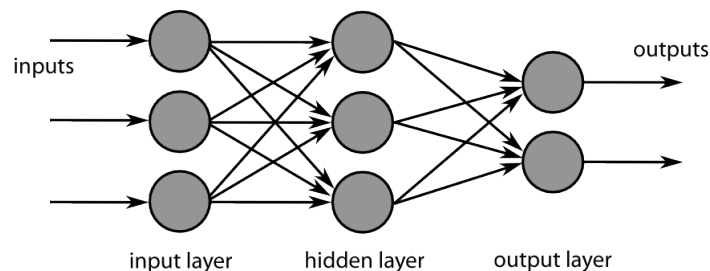
A neuronokból álló hálózatokat nevezzük neurális hálózatoknak. Ezekben minden neuron ugyanolyan, vagy hasonló műveleteket végez, a többi neurontól függetlenül, lokálisan. Tehát ezek a hálózatok olyan információfeldolgozó eszközök, amelyek párhuzamos, elosztott működésre, tanulásra képesek. Általában irányított gráffal reprezentáljuk őket. A neuronok a gráf csomópontjai, míg a gráf élei a kimenetek és bemenetek közötti kapcsolatot reprezentálják. Megvalósíthatók szoftveresen, hardveresen, vagy a kettő kombinációjaként is.

A neuronok három fajtáját különböztetjük meg:

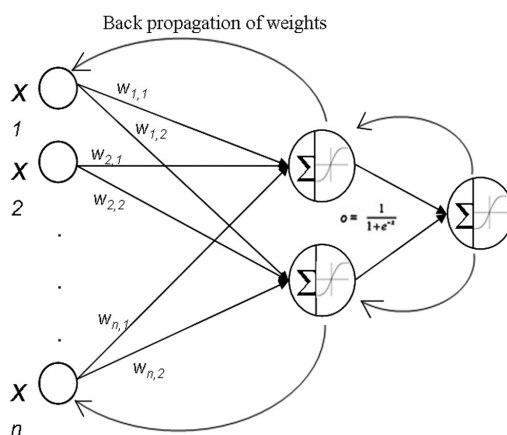
1. **bemeneti neuronok:** Egy bemenetű, egy kimenetű, buffer jellegű neuronok, jelfeldolgozó feladatuk nincs. Bemenetük a hálózat bemenete, kimenetük más neuronok meghajtására szolgál.
2. **rejtett neuronok:** Ezek a neuronok végzik a jelfeldolgozást. Kimenetük és bemenetük is más neuronokhoz csatlakozik.
3. **kimeneti neuronok:** A környezet felé továbbítják kimenetüket.

A neuronokat általában típusa alapján rétegekbe szervezzük. Ennek megfelelően beszélhetünk bemeneti rétegről, rejtett réteg(ek)ről és kimeneti rétegről.

A neuronhálózatokat az egyes neuronok közötti összeköttetési rendszer alapján két fő csoportba sorolhatjuk. Beszélhetünk előrecsatolt hálózatokról (2. ábra) és visszacsatolt hálózatokról. Akkor nevezzük egy neurális hálózatot visszacsatoltnak, ha a topológiáját reprezentáló irányított gráf tartalmaz hurkot. Ez esetben beszélhetünk globális és lokális visszacsatolásról.



2. ábra. Előrecsatolt neuron hálózatok felépítése.



3. ábra. A back-propagation algoritmus működése.

## 1.4. Back-propagation algoritmus

A back-propagation, teljes nevén „backward propagation of errors”, magyarul hiba-visszaterjesztési eljárás, egy tanulási algoritmus, melyet gyakran használnak a neurális hálózatokban. Ez egy felügyelt tanulási módszer, melynek szüksége van egy nagy adatbázisra a bemenetekkel és a kívánt kimenetekkel. Alkalmazása az előrecsatolt hálózatoknál a leghasznosabb. Használatához meg kell követelnünk, hogy a neuron hálózat réteges felépítésű, a neuron átviteli függvénye pedig deriválható legyen. Az algoritmusban a tanulás lényegében a hátrafelé terjedés folyamata, mely során minimalizálni kell az elvárt és a tényleges output vektor közötti négyzetes eltérést, Euklideszi távolságot.

Működése alapján két fázisra lehet osztani, terjedésre (propagation) és a súlyok frissítésére. A terjedés során a jel mind előre, mind hátra a szinapszisok és a neuronok szintjén lokális információk alapján terjed. A súlyok frissítése a neuron kimenetére visszaérkezett jel alapján történik (3. ábra).

## 2. Az algoritmus implementálása

Egy neurális háló beprogramozása sok időt vehet igénybe és fölösleges. Ezért egy, már készen levő neurális hálót fogunk alkalmazni a Barker kód teszteléséhez.

A szerző az alábbi megjegyzéssel tette közzé a C++ forráskódot:

```
// Written by: Paras Chopra
// Email: paras1987@gmail.com
// Web: www.paraschopra.com
// Comment: Use this code as you like , but please give me credit wherever i dese
```

A kód önmagában nem működött, először fordítási hibát kaptunk, majd azt kijavítva „segmentation fault” hibával szállt el. A hiba forrása a „Network” osztály destruktora:

```
\\ segmentation fault
~Network()
{
    delete Layers;
}
```

```
\\ helyesen
~Network()
{
    delete [] Layers;
}
```

Az eredeti program több más sebből is vérzett, de kisebb-nagyobb módosításokkal, átszervezéssel sikerült alap esetben a futási időt 25%-val lecsökkenteni. Természetesen másfajta, nem programozási módosításokra is szükség volt, hogy a célnak megfelelő neurális hálót kapjunk. A Barker kód 11 bitet feleltet meg 1 bitnek, ezért a neurális hálónknak 11 bemeneti neuronja, 11 rejtett neuronja és 1 kimenői neuronja van.

A végső forrás kód:

```
/*
 * neural.cpp
 *
 * Written by: Paras Chopra
 * Email: paras1987@gmail.com
 * Web: www.paraschopra.com
 * Comment: Use this code as you like , but please give me credit whenever I dese
 *
 * Improved version by Andras Mamenyak, Roland Bamli
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
```

```

*   along with this program. If not, see <http://www.gnu.org/licenses/>.
*
*   Changelog
*   - fixed build errors
*   - fixed segfault errors
*   - cleaner code
*   - improved runtime: 25% faster
*   - test the Barker 11 code
*/

#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

class Dendrite
{
public:
    double weight; // Weight of the neuron
    int points_to; // The index of the neuron of the next layer to which it points

    Dendrite(double weight = 0.0, int points_to = 0) : weight(weight), points_to(points_to) {}
};

class Neuron
{
public:
    Neuron(const int id = 0, const double value = 0.0, const double bias = 0.0) : id(id), value(value), bias(bias) {}

    ~Neuron()
    {
        delete[] dendrite;
    }

    /*
     * Set the dendrite from the neuron to given dendrite
     */
    void set_dendrite(const int n)
    {
        dendrite = new Dendrite[n];

        for (int i = 0; i < n; i++) // Initialize the dendrite to attach to next layer
            dendrite[i].points_to = i;
    }

    int id;

```

```

    double value, bias, delta;
    Dendrite *dendrite;
};

class Layer
{
public:
    Layer()
    {}

    ~Layer()
    {
        delete [] neuron;
    }

    void initialize(const int size)
    {
        neuron = new Neuron[size];
    }

    Neuron get_neuron(const int index) const
    {
        return neuron[index];
    }

    void set_neuron(Neuron neuron, const int index)
    {
        this->neuron[index] = neuron;
    }

    Neuron *neuron;
};

class Network
{
public:
    Network()
    {
        srand(time(NULL));
    }

    ~Network()
    {}

    /*
     * Set various parameters of the net
     */
    void set_data(const double learning_rate, const int layer[])
    {
        this->learning_rate = learning_rate;
    }

```

```

    for (int i = 0; i < 3; i++)
    {
        neuron_per_layer[i] = layer[i];
        this→layer[i].initialize(layer[i]);
// Initialize each layer with the specified size
    }

    randomize();
}

/*
 * The real test
 */
void test(const double input[], double output[])
{
    for (int i = 0; i < neuron_per_layer[0]; i++)
        layer[0].neuron[i].value = input[i];

    update_output();

    for (int i = 0; i < neuron_per_layer[2]; i++)
        output[i] = layer[2].neuron[i].value;
}

/*
 * The standard backprop learning algorithm
 *
 * For output layer:
 *  $\Delta = (Target - Actual) * Actual * (1 - Actual)$ 
 *
 * For hidden layer:
 *  $\Delta = Actual * (1 - Actual) * Sum(Weight\_from\_current\_to\_next \text{ AND } \Delta)$ 
 *
 *  $Weight += LearningRate * \Delta * Input$ 
 */
void train(const double input[], const double output[])
{
    double Actual, Delta;

    for (int i = 0; i < neuron_per_layer[0]; i++)
        layer[0].neuron[i].value = input[i];

    update_output();

    for (int i = 2; i > 0; i--) // Go from last layer to first layer
        for (int j = 0; j < neuron_per_layer[i]; j++)
        {
            Actual = layer[i].neuron[j].value;
// Actual value

```

```

        if(i == 2)  // Output layer
        {
            Delta = (output[j] - Actual) * Actual * (1.0 - Actual);
// Function to compute error
            layer[i].neuron[j].delta = Delta;
        }
        else  // Hidden layer
        {
            Delta = Actual * (1.0 - Actual) * sum_weight_delta(i);
        }

        if (i > 0)  // Input layer does not have a bias
            layer[i].neuron[j].bias += Delta*learning_rate;

        for (int k = 0; k < neuron_per_layer[i-1]; k++)
// Calculate the new weights
            layer[i-1].neuron[k].dendrite[j].weight += Delta*learning_rate*layer[i].neuron[j].delta;
    }
}

private:
/*
 * Randomize weights and biases
 */
void randomize()
{
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < neuron_per_layer[i]; j++)
        {
            if (i < 2)  // Last layer does not require weights
            {
                layer[i].neuron[j].set_dendrite(neuron_per_layer[i+1]); // Initialize dendrites
                for (int k = 0; k < neuron_per_layer[i+1]; k++)
                    layer[i].neuron[j].dendrite[k].weight = get_rand(); // Let weight be random
            }

            if (i > 0)  // First layer does not need biases
                layer[i].neuron[j].bias = get_rand();
        }
}

/*
 * Gives the output of the net
 */
void update_output()
{
    for (int i = 1; i < 3; i++)
        for (int j = 0; j < neuron_per_layer[i]; j++)

```



```

        {
            for (int k = 0; k < neuron_per_layer[i-1]; k++)
// Multiply and add all the inputs
                layer[i].neuron[j].value += layer[i-1].neuron[k].value*layer[i-1].neu

                layer[i].neuron[j].value += layer[i].neuron[j].bias;
// Add bias
                layer[i].neuron[j].value = limiter(layer[i].neuron[j].value);
// Squash that value
        }
    }

    /*
     * Sigmoid activation function
     */
    double limiter(const double x) const
    {
        return 1.0/(1.0 + exp(-x));
    }

    double get_rand() const
    {
        return -1.0 + ((double) rand())/RAND_MAX)*2.0;
    }

    /*
     * Calculate sum of weights * delta. Used in back prop.
     */
    double sum_weight_delta(const int Nlayer) const
    {
        double result = 0.0;

        for (int i = 0; i < neuron_per_layer[Nlayer+1]; i++)
// Go through all the neuron in the next layer
            result += layer[Nlayer].neuron[Nlayer].dendrite[i].weight*layer[Nlayer+1]

        return result;
    }

    double learning_rate;
    Layer layer[3];
    int neuron_per_layer[3];
};

int main()
{
    const int Niter = 10000;
    const int Nneuron = 11;
    const int Ntrain = 256;
    const int Ntest = pow(2, Nneuron);

```

```

    const int layer[3] = {Nneuron, Nneuron, 1};
    // input, hidden, output

    Network network;
    network.set_data(0.1, layer);

    std::cout << "Start_training.\n\n";

    double train_input[Ntrain][layer[0]];
    double train_output[Ntrain][layer[2]];

    // 11100010010 - 1811
    train_input[0][0] = train_input[0][1] = train_input[0][2] = train_input[0][6]
    train_input[0][3] = train_input[0][4] = train_input[0][5] = train_input[0][7]

    for (int i = 1; i < Ntrain; i++)
        for (int j = 0; j < layer[0]; j++)
            train_input[i][j] = (double) (rand()%2);

    for (int i = 0; i < Ntrain; i++)
    {
        if (train_input[i][0] + train_input[i][1] + train_input[i][2] + train_input
            train_input[i][3] + train_input[i][4] + train_input[i][5] + train_input
            train_output[i][0] = 1;
        else
            train_output[i][0] = 0;
    }

    std::cout << "Number_of_training_iterations:_ " << Niter;

    for (int i = 0; i < Niter; i++)
        for (int j = 0; j < Ntrain; j++)
            network.train(train_input[j], train_output[j]);

    std::cout << "\nEnd_training.\n";

    std::cout << "\nStart_testing.\n";

    double test_input[Ntest][layer[0]];
    double test_output[layer[2]];
    int db = 0;

    for (int i = 0; i < Ntest; i++)
    {
        int tmp = i;
        int j = Nneuron - 1;

        while (tmp > 0)

```

```

    {
        test_input[i][j] = tmp%2;
        tmp /= 2;
        j--;
    }

    while (j > 0)
    {
        test_input[i][j] = 0;
        j--;
    }
}

for (int i = 0; i < Ntest; i++)
{
    network.test(test_input[i], test_output);

    if (test_output[0] > 0.5)
    {
        db++;
        std::cout << "\nCase_number:_ " << db << "\n";

        std::cout << "Input:_";
        for (int j = 0; j < layer[0]; j++)
            std::cout << test_input[i][j];
        std::cout << "_(" << i+1 << ") \n";

        std::cout << "Output:_";
        for (int j = 0; j < layer[2]; j++)
            std::cout << test_output[j];
        std::cout << "\n";
    }
}

std::cout << "\nEnd_testing.\n";

std::cout << "\nNumber_of_positive_output:_ " << db << "\n";

return 0;
}

```

### 3. A program futtatása és a kimenet értelmezése

A program fordítása és futtatása az alábbi módon végezhető el:

```

andras@G53SW:~/Programs/Neural$ g++ neural.cpp -o neural
andras@G53SW:~/Programs/Neural$ ./neural

```

Start training.

Number of training iterations: 10000  
End training.

Start testing.

Case number: 1  
Input: 01100010010 (787)  
Output: 0.870079

Case number: 2  
Input: 01100010011 (788)  
Output: 0.578244

Case number: 3  
Input: 10100010010 (1299)  
Output: 0.807933

Case number: 4  
Input: 11000010010 (1555)  
Output: 0.946152

Case number: 5  
Input: 11000010011 (1556)  
Output: 0.759956

Case number: 6  
Input: 11001010010 (1619)  
Output: 0.516823

Case number: 7  
Input: 11100000010 (1795)  
Output: 0.766867

Case number: 8  
Input: 11100010000 (1809)  
Output: 0.725297

Case number: 9  
Input: 11100010010 (1811)  
Output: 0.987942

Case number: 10  
Input: 11100010011 (1812)  
Output: 0.954111

Case number: 11  
Input: 11100010100 (1813)  
Output: 0.762585

Case number: 12  
Input: 11100010110 (1815)  
Output: 0.798953

Case number: 13  
Input: 11100110010 (1843)  
Output: 0.725913

Case number: 14  
Input: 11101010010 (1875)  
Output: 0.823665

Case number: 15  
Input: 11110010010 (1939)  
Output: 0.704915

End testing.

Number of positive output: 15

A neurális hálót

```
const int Niter = 10000;
```

iteráción keresztül „tanítottuk”, mindegyik esetben ugyanazt a

```
const int Ntrain = 256;
```

random bemeneti adatot és a hozzájuk tartozó kimeneti értéket tápláljuk a neurális hálóba. Ez alapján állítódnak be az egyes neuronokhoz tartozó súlyok, amik kezdetben random értékek voltak. Ezután az éles tesztben ráengedjük mind a  $2^{11}$  esetet és amint a fenti kimenetben látható, csupán 15 esetben lépte át a kimenet értéke a 0.5-öt. A várt 11100010010 (1811)-as értékre kaptuk a legjobb eredményt, 0.987942-t. Ebből jól látható, hogy a neurális hálónk megfelelően működik, megtanulta melyik a Barker kód, és csak nagyjából 1 bitben különböző bemenetekre ad magasabb kimeneti értéket.

## 4. Az algoritmus finomhangolása

Sok állítható paraméterrel rendelkezik az algoritmusunk, ezért számos beállítással kipróbáltuk. A cél az volt, hogy a legnagyobb pontosságot érjük el viszonylag rövid futási időn belül. Ha az iterációk számát növeltük, lineárisan növekedett a futási idő is, viszont az eredmény nem lett sokkal pontosabb. Ugyanez érvényes a bemeneti adatok számára is. A legnagyobb különbséget a rejtett neuronok számának növelése jelentette, így értük el optimális futási idő alatt

```
real      0m7.484 s  
user      0m7.463 s  
sys       0m0.014 s
```

a megfelelő pontosságot.

## 5. Konklúzió

A projekt során a ma népszerű kutatási területnek számító neurális hálózattal foglalkoztunk. Megismerkedtünk az alapvető felépítésével és működésével ezeken a gépi tanulást végző rendszereknek, valamint sikerült a Barker 11 kódot helyesen tesztelő programot készítenünk.

A program letölthető a <https://github.com/mamenyaka/Neural> github tárolóból.