# Artificial Neural Networks

Abdullatif alShriaf

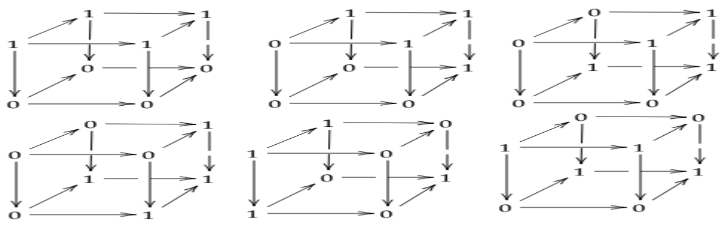
September 2018

## 1 Number of 3-dimensional Boolean Functions

We could easily count the number of this kind of functions using The Fundamental Counting Principle to be $2^{2^3} = 256$

## 2 Number of Symmetries of 3-dimensional Boolean Functions with Four Ones

With total of 70 possible functions, we need to divide them into symmetries, or equivalence classes. All functions in a symmetry result from rotation(s) and/or reflection(s) on any one of them. We recognize 6 total equivalence classes.



## 3 Linearly Separable 3-dimensional Functions

Using the same previous method by dividing each 3-dimensional function into symmetries, and keeping a count on how many functions each symmetry have, one could count the number of linearly separable functions by examining a candidate from each symmetry and then adding the number of the function in a symmetry if the candidate *is* linearly separable; We also note that the number of linearly separable functions with 0 ones, is equivalent to that with 8 ones, and those with 1 one to that of 7 ones and so on.

$$\mathrm{L} = \Sigma_{n=0}^{n=8} \mathrm{L}_n$$

$$\mathrm{L} = 1 + 8 + 18 + 12 + 26 + 12 + 18 + 8 + 1 = 104$$

# main.cpp

```cpp
#include <iostream>
#include <random>
#include <math.h>

using namespace std;


double generateRandom(double, double);

double activationFunction(double field);


double tanhDerivative(double val);

int sgn(double x){
    return x>=0?1:-1;
}

#define ETA 0.02
#define N 4


double weights[N] = {0};
double threshold;

int input[16][N] = {
        {- 1, - 1, - 1, - 1},
        {1,   - 1, - 1, - 1},
        {- 1, 1,   - 1, - 1},
        {- 1, - 1, 1,   - 1},
        {- 1, - 1, - 1, 1},
        {1,   1,   - 1, - 1},
        {1,   - 1, 1,   - 1},
        {1,   - 1, - 1, 1},
        {- 1, 1,   1,   - 1},
        {- 1, 1,   - 1, 1},
        {- 1, - 1, 1,   1},
        {1,   1,   1,   - 1},
        {1,   1,   - 1, 1},
        {1,   - 1, 1,   1},
        {- 1, 1,   1,   1},
        {1,   1,   1,   1}
};


int targets[][16] =
        {
                {- 1, - 1, - 1, 1,   - 1, 1,   1,   1,   1,   1,   1,   - 1, 1,   - 1, - 1, - 1},

                {- 1, 1,   1,   - 1, - 1, 1,   - 1, - 1, - 1, - 1, 1,   1,   - 1, 1,   - 1, - 1},
                {1,   - 1, 1,   - 1, - 1, 1,   - 1, - 1, - 1, - 1, - 1, - 1, 1,   - 1, - 1, 1},
                {- 1, 1,   - 1, - 1, - 1, - 1, - 1, 1,   - 1, - 1, - 1, - 1, - 1, 1,   - 1, - 1},
                {1,   1,   1,   - 1, - 1, 1,   - 1, - 1, 1,   1,   - 1, 1,   1,   - 1, - 1, - 1},
                {1,   1,   - 1, 1,   - 1, - 1, 1,   - 1, 1,   - 1, 1,   1,   - 1, 1,   1,   - 1},
                {1, 1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1,   1}

        };

double getField(const int pattern[N]) {
    double field = 0.0;
    for (int i = 0; i < N; i ++) {
        field += weights[i] * pattern[i];
    }
    return (field - threshold);
}

double predict(const int pattern[N]) {
    return activationFunction(getField(pattern));
}
```

```cpp
int testfunction(int target[16]) {


    int best_acc = 0;

    for (int i = 0; i < N; i ++) {
        weights[i] = generateRandom(- 0.2, 0.2);
    }
    threshold = generateRandom(- 1, 1);

    for (int t = 0; t <= 100000; t ++) {


        double error;

        int mu = (int) (generateRandom(0, 16));

        double field = getField(input[mu]);
        double prediction = activationFunction(field);

        error = tanhDerivative(field) * (target[mu] - prediction);

        for (int j = 0; j < N; j ++) {
            weights[j] += ETA * error * input[mu][j];
        }
        threshold += ETA * error;

        double acc = 0;
        for (int m = 0; m < 16; m ++) {
            auto pred = predict(input[m]);
            if (sgn(pred)==target[m])
                acc ++;
        }
        acc = acc;
        if (acc>best_acc)
            best_acc=acc;
        if (best_acc==16)
            break;
    }


    return best_acc;

}

int main() {

    int best[]={0,0,0,0,0,0,0};

    for (int experiment = 0; experiment<10;experiment++) {
        for (char function = 'A'; function <= 'F'; function ++) {
            int record = testfunction(targets[function - 'A']);
            if (record>best[function-'A']){
                best[function-'A'] = record;
            }
        }
    }


    for (char function = 'A'; function <= 'F'; function ++) {
        cout << "Function " << function << " :"<<100*best[function-'A']/16.0<<"% accuracy"<<endl;

    }

}


double generateRandom(double a, double b) {
    std::random_device rd;
    std::uniform_real_distribution<> uni(a, b);
    return uni(rd);
}

double activationFunction(double field) {
    return tanh(field);
}

double tanhDerivative(double val) {
    return (1.0 - pow(tanh(val), 2.0));
}
```

# DataSet.cpp

```
//
// Created by latiif on 9/24/18.
//

#include <string.h>
#include "DataSet.h"
#include "fstream"

DataSet::DataSet(std::string filename) {


    std::ifstream ifs (filename, std::ifstream::in);

    this->total  = 0;
    int nLines = 0;
    while (!ifs.eof())
    {

        double x,y;
        double t;
        char temp;

        ifs>>x>>temp>>y>>temp>>t;


        targets[nLines] = (int)t;
        samples[nLines][0] = x;
        samples[nLines][1] = y;


        nLines++;
    }
    this->total = nLines - 1;

}
```

# DataSet.h

```
//
// Created by latiif on 9/24/18.
//

#ifndef TRAINING_TRAININGSET_H
#define TRAINING_TRAININGSET_H

#include "string"

#define MAX_DATASET_LENGTH 15000

class DataSet {
public:
    double samples[MAX_DATASET_LENGTH][2];
    int targets[MAX_DATASET_LENGTH];

    DataSet(std::string filename);

    int total;
};


#endif //TRAINING_TRAININGSET_H
```

# main.cpp

```cpp
#include <iostream>
#include "DataSet.h"
#include "algorithm"
#include "Network.h"
#include "Misc.h"
#include <cstdlib>


#define V 5000
#define ETA 0.0099

int main(int argc, char **argv) {
    double classificationError = 100.0;

    auto train = new DataSet("../../training_set.csv");
    auto validation = new DataSet("../../validation_set.csv");

    int m1, m2;

    m1 = atoi(argv[1]);
    m2 = atoi(argv[2]);

    auto T = train->total;
    auto EPOCHS = T * 100;

    double bestClassification = 100.0;
    double lastClassification = 0.0;

    auto network = new Network(m1, m2, ETA);
    Network *bestNetwork = nullptr;

    int count = 0;

    while (count ++ < EPOCHS) {

        int mu = Misc::generateRandomUniform(0, T - 1);


        network->train(train->samples[mu], 2, train->targets[mu]);

        if (count % T == 0) {
            int correct = 0;
            int pValue = V;
            classificationError = 0.0;
            for (int i = 0; i < pValue; i ++) {
                auto guess = network->predict(validation->samples[i], 2);
                auto target = validation->targets[i];
                correct += (guess == target ? 1 : 0);
                classificationError += abs((int) (guess - target));
            }

            classificationError = (classificationError * 1.0) / (2 * pValue);


            if (classificationError <= 1) {

                if (classificationError < bestClassification) {
                    bestClassification = classificationError;
                    delete bestNetwork;
                    bestNetwork = network->makeCopy();
                }

                if (classificationError == lastClassification) {
                    break;
                }
                lastClassification = classificationError;
            }
        }
    }

    if (bestClassification < 0.12 && bestNetwork != nullptr) {
        bestNetwork->saveCSV();
    }



    /// Since the problem is 2D, we can ask the network to imagine what it has learned
    if (bestNetwork != nullptr) {
        int c = 0;
        for (double x = - 1; x <= 1; x += 0.02) {
            for (double y = - 1; y <= 1; y += 0.02) {
                c ++;
                auto input = new Inputs(2);
                (*input)[0] = x;
                (*input)[1] = y;
                auto res = bestNetwork->predict(input);
                std::cout << (res == - 1 ? 0 : 1);
                if (c % 100 == 0) std::cout << std::endl;
                delete (input);
            }
        }
        std::cout << "Best:\t" << bestClassification << "\tETA:\t" << ETA << '\t' << m1
                  << '\t' << m2
                  << std::endl;
    }


    delete network;
    delete bestNetwork;


    return 0;
}
```

# Misc.cpp

```
//
// Created by latiif on 2018-09-24.
//

#include "Misc.h"
#include <random>

double Misc::generateRandomNormal(double mean, double variance) {
    std::random_device rd;
    std::normal_distribution<> uni(mean, variance);
    return uni(rd);

}

double Misc::max(double a, double b) {
    return (a > b ? a : b);
}

int Misc::generateRandomUniform(int a, int b) {
    std::random_device rd;
    std::uniform_int_distribution<int> uni(a, b);
    return uni(rd);

}
```

# Misc.h

```
//
// Created by latiif on 2018-09-24.
//

#ifndef TRAINING_MISC_H
#define TRAINING_MISC_H


class Misc {

public:

    static double generateRandomNormal(double a, double b);

    static int generateRandomUniform(int a, int b);

    static double max(double a, double b);
};


#endif //TRAINING_MISC_H
```

# Network.cpp

```cpp
//
// Created by latiif on 9/24/18.
//

#include "Network.h"
#include "vector"
#include <math.h>
#include "random"
#include "Misc.h"
#include <fstream>

Network::Network(int M1, int M2, double learningRate) {

    this->w1 = new WeightMatrix();
    this->w2 = new WeightMatrix();
    this->w3 = new WeightMatrix();

    this->setSize(w1, M1, 2);
    this->setSize(w2, M2, M1);
    this->setSize(w3, 1, M2);

    th1 = new Threshold();
    th1->resize(M1);
    initThreshold(M1, th1);


    th2 = new Threshold();
    th2->resize(M2);
    initThreshold(M2, th2);

    th3 = new Threshold();
    th3->resize(1);
    initThreshold(1, th3);

    this->learningRate = learningRate;

}

void Network::setSize(WeightMatrix *weightMatrix, int d1, int d2) {
    weightMatrix->resize(d1);
    for (int i = 0; i < d1; i ++) (*weightMatrix)[i].resize(d2);

    for (int i = 0; i < d1; i ++)
        for (int j = 0; j < d2; j ++)
            (*weightMatrix)[i][j] = Misc::generateRandomNormal(0, 1);

}

double Network::activationFunction(double field) {
    return tanh(field);
}

double Network::activationFunctionDerivative(double val) {
    return (1 - pow(tanh(val), 2.0));
}

DetailedResult *
Network::calculateValues(const Inputs *inputs, const WeightMatrix *w, const Threshold *thresholds, int nInputs,
                         int nOutputs) {
    auto result = new DetailedResult(new Activations(nOutputs), new Fields(nOutputs));

    for (int neuron = 0; neuron < nOutputs; neuron ++) {

        double field = 0.0;
        for (int i = 0; i < nInputs; i ++) {
            field += (*w)[neuron][i] * (*inputs)[i];
        }
        (*(*result).first)[neuron] = activationFunction(field - (*thresholds)[neuron]);
        (*(*result).second)[neuron] = field;
    }

    return result;
}

double Network::predict(Inputs *pattern) {

    auto activations = calculateValues(pattern, w1, th1, 2, w1->size())->first;

    activations = calculateValues(activations, w2, th2, activations->size(), w2->size())->first;

    activations = calculateValues(activations, w3, th3, activations->size(), w3->size())->first;

    return sgn((*activations)[0]);
}

int Network::sgn(double val) {
    return val >= 0 ? 1 : - 1;
}

double Network::predict(double *pattern, int size) {
    auto inputPattern = new Inputs(size);

    for (int i = 0; i < size; i ++) (*inputPattern)[i] = pattern[i];

    return sgn(predict(inputPattern));
}


void Network::initThreshold(int size, Threshold *threshold) {
    for (int i = 0; i < size; i ++)
        (*threshold)[i] = 0;
}

double Network::train(Inputs *pattern, int target) {


    auto layerOne = calculateValues(pattern, w1, th1, 2, w1->size());

    auto layerTwo = calculateValues(layerOne->first, w2, th2, layerOne->first->size(), w2->size());

    auto outputLayer = calculateValues(layerTwo->first, w3, th3, layerTwo->first->size(), w3->size());


    Errors *errors[3];

    /// Train the output layer w3,th3
    errors[2] = new Errors(1);
    for (int i = 0; i < 1; i ++) {
        double error;
        double field = outputLayer->second->at(i);
        double prediction = outputLayer->first->at(i);

        error = activationFunctionDerivative(field) * (target - prediction);

        errors[2]->at(i) = error;

    }

    /// Train the second hidden layer w2, th2
    errors[1] = new Errors(th2->size());

    for (int j = 0; j < th2->size(); j ++) {
        double error = 0.0;
        double field = layerTwo->second->at(j);


        for (int i = 0; i < th3->size(); i ++) {
            error += errors[2]->at(i) * (*w3)[i][j] * activationFunctionDerivative(field);
        }
        errors[1]->at(j) = error;

    }


    /// Train the first hidden layer w1, th1
    errors[0] = new Errors(th1->size());

    for (int j = 0; j < th1->size(); j ++) {
        double error = 0.0;
        double field = layerOne->second->at(j);


        for (int i = 0; i < th2->size(); i ++) {
            error += errors[1]->at(i) * (*w2)[i][j] * activationFunctionDerivative(field);
        }
        errors[0]->at(j) = error;
    }


    /// Update weights for the first layer
    updateWeightMatrixAndTheta(w1, th1, errors[0], pattern);
    updateWeightMatrixAndTheta(w2, th2, errors[1], layerOne->first);
    updateWeightMatrixAndTheta(w3, th3, errors[2], layerTwo->first);

    //Free up memory used
    delete errors[0];
    delete errors[1];
    delete errors[2];
    delete layerOne;
    delete layerTwo;
    delete outputLayer;

    return 0;
}
```

```
double Network::train(double *pattern, int size, int target) {
    auto inputPattern = new Inputs(size);

    for (int i = 0; i < size; i ++) (*inputPattern)[i] = pattern[i];

    return train(inputPattern, target);
}

Network::~Network() {
    delete w1;
    delete w2;
    delete w3;

    delete th1;
    delete th2;
    delete th3;
}

void
Network::updateWeightMatrixAndTheta(WeightMatrix *w, Threshold *threshold, Errors *error, Activations *activations) {
    for (int i = 0; i < w->size(); i ++) {
        for (int j = 0; j < w->at(i).size(); j ++) {
            w->at(i)[j] += learningRate * error->at(i) * activations->at(j);
        }

        threshold->at(i) -= learningRate * error->at(i);
    }


}

Network *Network::makeCopy() {
    auto res = new Network(th1->size(), th2->size(), learningRate);

    res->w1 = new WeightMatrix(*w1);
    res->w2 = new WeightMatrix(*w2);
    res->w3 = new WeightMatrix(*w3);

    res->th1 = new Threshold(*th1);
    res->th2 = new Threshold(*th2);
    res->th3 = new Threshold(*th3);


    return res;
}

void Network::saveCSV() {

    writeMatrix(w1, "w1.csv");
    writeMatrix(w2, "w2.csv");
    writeMatrix(w3, "w3.csv");

    writeThreshold(th1, "t1.csv");
    writeThreshold(th2, "t2.csv");
    writeThreshold(th3, "t3.csv");
}

void Network::writeMatrix(WeightMatrix *w, const std::string filename) {
    std::ofstream output;
    output.open(filename);

    for (int row = 0; row < w->size(); row ++) {
        for (int col = 0; col < w->at(row).size(); col ++) {
            output << w->at(row)[col];
            if (col == w->at(row).size() - 1) { output << std::endl; }
            else { output << " , "; }
        }
    }

    output.close();
}

void Network::writeThreshold(Threshold *th, const std::string filename) {
    std::ofstream output;
    output.open(filename);

    for (int row = 0; row < th->size(); row ++) {
        output << th->at(row) << std::endl;
    }

    output.close();
}
```

# Network.h

```
//
// Created by latiif on 9/24/18.
//

#ifndef TRAINING_NETWORK_H
#define TRAINING_NETWORK_H

#include "vector"
#include <string>

typedef std::vector<std::vector<double>> WeightMatrix;
typedef std::vector<double> Threshold;
typedef std::vector<double> Activations;
typedef std::vector<double> Fields;

typedef std::vector<double> Errors;

typedef std::pair<Activations *, Fields *> DetailedResult;

typedef std::vector<double> Inputs;

class Network {
private:

    double learningRate;

    void updateWeightMatrixAndTheta(WeightMatrix *w, Threshold *threshold, Errors *error, Activations *activations);

    void setSize(WeightMatrix *weightMatrix, int d1, int d2);

    double activationFunction(double field);

    inline void initThreshold(int size, Threshold *threshold);

    double activationFunctionDerivative(double val);

    int sgn(double);

    DetailedResult *calculateValues(const Inputs *, const WeightMatrix *, const Threshold *, int nInputs, int nOutputs);


    void writeMatrix(WeightMatrix *w, const std::string filename);

    void writeThreshold(Threshold *th, const std::string filename);

public:
    WeightMatrix *w1, *w2, *w3;
    Threshold *th1, *th2, *th3;

    Network(int M1, int M2, double eta);

    ~Network();

    Network *makeCopy();

    double predict(Inputs *pattern);

    double predict(double pattern[], int size);

    double train(double pattern[], int size, int target);

    double train(Inputs *pattern, int target);

    void saveCSV();
};


#endif //TRAINING_NETWORK_H
```