// Other techniques for learning

class ActivationFunction {

constructor(func, dfunc) {

this.func = func;

this.dfunc = dfunc;

}

}

let sigmoid = new ActivationFunction(

x => 1 / (1 + Math.exp(-x)),

y => y \* (1 - y)

);

let tanh = new ActivationFunction(

x => Math.tanh(x),

y => 1 - (y \* y)

);

class NeuralNetwork {

/\*

\* if first argument is a NeuralNetwork the constructor clones it

\* USAGE: cloned\_nn = new NeuralNetwork(to\_clone\_nn);

\*/

constructor(in\_nodes, hid\_nodes, out\_nodes) {

if (in\_nodes instanceof NeuralNetwork) {

let a = in\_nodes;

this.input\_nodes = a.input\_nodes;

this.hidden\_nodes = a.hidden\_nodes;

this.output\_nodes = a.output\_nodes;

this.weights\_ih = a.weights\_ih.copy();

this.weights\_ho = a.weights\_ho.copy();

this.bias\_h = a.bias\_h.copy();

this.bias\_o = a.bias\_o.copy();

} else {

this.input\_nodes = in\_nodes;

this.hidden\_nodes = hid\_nodes;

this.output\_nodes = out\_nodes;

this.weights\_ih = new Matrix(this.hidden\_nodes, this.input\_nodes);

this.weights\_ho = new Matrix(this.output\_nodes, this.hidden\_nodes);

this.weights\_ih.randomize();

this.weights\_ho.randomize();

this.bias\_h = new Matrix(this.hidden\_nodes, 1);

this.bias\_o = new Matrix(this.output\_nodes, 1);

this.bias\_h.randomize();

this.bias\_o.randomize();

}

// TODO: copy these as well

this.setLearningRate();

this.setActivationFunction();

}

predict(input\_array) {

// Generating the Hidden Outputs

let inputs = Matrix.fromArray(input\_array);

let hidden = Matrix.multiply(this.weights\_ih, inputs);

hidden.add(this.bias\_h);

// activation function!

hidden.map(this.activation\_function.func);

// Generating the output's output!

let output = Matrix.multiply(this.weights\_ho, hidden);

output.add(this.bias\_o);

output.map(this.activation\_function.func);

// Sending back to the caller!

return output.toArray();

}

setLearningRate(learning\_rate = 0.1) {

this.learning\_rate = learning\_rate;

}

setActivationFunction(func = sigmoid) {

this.activation\_function = func;

}

train(input\_array, target\_array) {

// Generating the Hidden Outputs

let inputs = Matrix.fromArray(input\_array);

let hidden = Matrix.multiply(this.weights\_ih, inputs);

hidden.add(this.bias\_h);

// activation function!

hidden.map(this.activation\_function.func);

// Generating the output's output!

let outputs = Matrix.multiply(this.weights\_ho, hidden);

outputs.add(this.bias\_o);

outputs.map(this.activation\_function.func);

// Convert array to matrix object

let targets = Matrix.fromArray(target\_array);

// Calculate the error

// ERROR = TARGETS - OUTPUTS

let output\_errors = Matrix.subtract(targets, outputs);

// let gradient = outputs \* (1 - outputs);

// Calculate gradient

let gradients = Matrix.map(outputs, this.activation\_function.dfunc);

gradients.multiply(output\_errors);

gradients.multiply(this.learning\_rate);

// Calculate deltas

let hidden\_T = Matrix.transpose(hidden);

let weight\_ho\_deltas = Matrix.multiply(gradients, hidden\_T);

// Adjust the weights by deltas

this.weights\_ho.add(weight\_ho\_deltas);

// Adjust the bias by its deltas (which is just the gradients)

this.bias\_o.add(gradients);

// Calculate the hidden layer errors

let who\_t = Matrix.transpose(this.weights\_ho);

let hidden\_errors = Matrix.multiply(who\_t, output\_errors);

// Calculate hidden gradient

let hidden\_gradient = Matrix.map(hidden, this.activation\_function.dfunc);

hidden\_gradient.multiply(hidden\_errors);

hidden\_gradient.multiply(this.learning\_rate);

// Calcuate input->hidden deltas

let inputs\_T = Matrix.transpose(inputs);

let weight\_ih\_deltas = Matrix.multiply(hidden\_gradient, inputs\_T);

this.weights\_ih.add(weight\_ih\_deltas);

// Adjust the bias by its deltas (which is just the gradients)

this.bias\_h.add(hidden\_gradient);

// outputs.print();

// targets.print();

// error.print();

}

serialize() {

return JSON.stringify(this);

}

static deserialize(data) {

if (typeof data == 'string') {

data = JSON.parse(data);

}

let nn = new NeuralNetwork(data.input\_nodes, data.hidden\_nodes, data.output\_nodes);

nn.weights\_ih = Matrix.deserialize(data.weights\_ih);

nn.weights\_ho = Matrix.deserialize(data.weights\_ho);

nn.bias\_h = Matrix.deserialize(data.bias\_h);

nn.bias\_o = Matrix.deserialize(data.bias\_o);

nn.learning\_rate = data.learning\_rate;

return nn;

}

// Adding function for neuro-evolution

copy() {

return new NeuralNetwork(this);

}

// Accept an arbitrary function for mutation

mutate(func) {

this.weights\_ih.map(func);

this.weights\_ho.map(func);

this.bias\_h.map(func);

this.bias\_o.map(func);

}

}

// let m = new Matrix(3,2);

class Matrix {

constructor(rows, cols) {

this.rows = rows;

this.cols = cols;

this.data = Array(this.rows).fill().map(() => Array(this.cols).fill(0));

}

copy() {

let m = new Matrix(this.rows, this.cols);

for (let i = 0; i < this.rows; i++) {

for (let j = 0; j < this.cols; j++) {

m.data[i][j] = this.data[i][j];

}

}

return m;

}

static fromArray(arr) {

return new Matrix(arr.length, 1).map((e, i) => arr[i]);

}

static subtract(a, b) {

if (a.rows !== b.rows || a.cols !== b.cols) {

console.log('Columns and Rows of A must match Columns and Rows of B.');

return;

}

// Return a new Matrix a-b

return new Matrix(a.rows, a.cols)

.map((\_, i, j) => a.data[i][j] - b.data[i][j]);

}

toArray() {

let arr = [];

for (let i = 0; i < this.rows; i++) {

for (let j = 0; j < this.cols; j++) {

arr.push(this.data[i][j]);

}

}

return arr;

}

randomize() {

return this.map(e => Math.random() \* 2 - 1);

}

add(n) {

if (n instanceof Matrix) {

if (this.rows !== n.rows || this.cols !== n.cols) {

console.log('Columns and Rows of A must match Columns and Rows of B.');

return;

}

return this.map((e, i, j) => e + n.data[i][j]);

} else {

return this.map(e => e + n);

}

}

static transpose(matrix) {

return new Matrix(matrix.cols, matrix.rows)

.map((\_, i, j) => matrix.data[j][i]);

}

static multiply(a, b) {

// Matrix product

if (a.cols !== b.rows) {

console.log('Columns of A must match rows of B.');

return;

}

return new Matrix(a.rows, b.cols)

.map((e, i, j) => {

// Dot product of values in col

let sum = 0;

for (let k = 0; k < a.cols; k++) {

sum += a.data[i][k] \* b.data[k][j];

}

return sum;

});

}

multiply(n) {

if (n instanceof Matrix) {

if (this.rows !== n.rows || this.cols !== n.cols) {

console.log('Columns and Rows of A must match Columns and Rows of B.');

return;

}

// hadamard product

return this.map((e, i, j) => e \* n.data[i][j]);

} else {

// Scalar product

return this.map(e => e \* n);

}

}

map(func) {

// Apply a function to every element of matrix

for (let i = 0; i < this.rows; i++) {

for (let j = 0; j < this.cols; j++) {

let val = this.data[i][j];

this.data[i][j] = func(val, i, j);

}

}

return this;

}

static map(matrix, func) {

// Apply a function to every element of matrix

return new Matrix(matrix.rows, matrix.cols)

.map((e, i, j) => func(matrix.data[i][j], i, j));

}

print() {

console.table(this.data);

return this;

}

serialize() {

return JSON.stringify(this);

}

static deserialize(data) {

if (typeof data == 'string') {

data = JSON.parse(data);

}

let matrix = new Matrix(data.rows, data.cols);

matrix.data = data.data;

return matrix;

}

} // end of class

if (typeof module !== 'undefined') {

module.exports = Matrix;

}